Intel® Integrated Performance Primitives for Intel® Architecture

Reference Manual

Volume 3:
Small Matrices

Document Number: A68761-008
World Wide Web: http://developer.intel.com

Version	Version Information	Date
-2001	Documents Intel® Integrated Performance Primitives (Intel® IPP) release 2.0 beta.	08/ 2001
-3001	Documents Intel IPP 3.0 beta release.	06/ 2002
-3002	Documents Intel IPP 3.0 gold.	11/ 2002
-4001	Documents Intel IPP 4.0 beta release	05/2003
-005	Documents Intel IPP 4.1 beta release	05/2004
-006	Documents Intel IPP 5.0 beta release. Full revision of API has been implemented.	03/2005
-007	Documents Intel IPP 5.0 gold release. Added code examples.	08/2005
-008	Documents Intel IPP 5.1 gold release. Several code examples added.	02/2006

Contents

Overview 1

This manual describes the structure, operation, and functions of the Intel® Integrated Performance Primitives (Intel® IPP) for small matrices. This is the third volume of the Intel IPP Reference Manual, which also comprises descriptions of Intel IPP for signal processing (volume 1), Intel IPP for image and video processing (volume 2), and Intel IPP for cryptography (volume 4). The Intel IPP software package supports many functions whose performance can be significantly enhanced on the Intel® Architecture (IA), particularly using the MMX™ technology and Streaming SIMD Extensions.

The Intel IPP for small matrices is a cross-platform software layer optimized for IA-32 and Intel® Itanium® architecture and running on Microsoft* Windows* and Linux* operating systems.

This manual provides detailed description of Intel IPP functions developed for operations on small matrices.

This chapter introduces the Intel IPP matrix operating software and explains the organization of this manual.

About This Software

The Intel IPP software enables to take advantage of the parallelism of the single-instruction, multiple data (SIMD) instructions that comprise the core of the MMX technology and Streaming SIMD Extensions.

Hardware and Software Requirements

Intel IPP for Intel architecture software runs on personal computers that are based on IA-processors or Itanium® architecture-based processors and running on Microsoft* Windows* 2000, Windows* ME, Windows* XP, or Linux*. Intel IPP integrates into the customer's application or library written in C or C++.

About This Manual

This manual provides a background for matrix operating concepts used in the Intel IPP software as well as detailed description of the respective Intel IPP functions. The Intel IPP functions are combined in groups by their functionality. Each group of functions is described in a separate chapter.

Manual Organization

This manual contains the following chapters:

Chapter 1	Overview. Introduces Intel IPP for matrix operations, provides information on manual organization, and explains notational conventions.
Chapter 2	Getting Started. Explains basic concepts underlying Intel IPP functions used for matrix operations and describes the supported data layout and operation modes.
Chapter 3	Utility Functions. Describes functions used to copy an object of any type to another object of any type, extract Regions of Interest (ROI), and initialize matrices.
Chapter 4	Vector Algebra Functions. Describes Intel IPP functions used to add, subtract, scale vectors, and perform other operations included in the vector algebra group.
Chapter 5	Matrix Algebra Functions. Describes Intel IPP functions used to add, subtract, and scale matrices, obtain matrix-vector and matrix-matrix products, and perform other operations of matrix algebra.
Chapter 6	Linear System Solution Functions. Describes functions used for LU decomposition and Cholesky decomposition for solving the system of linear equations by back substitution.

Chapter 7	Least Squares Problem Functions. Describes Intel IPP functions used for computing the matrix QR-decomposition and solving the least squares problem.
Chapter 8	Eigenvalue Problem Functions. Describes Intel IPP functions used for computing eigenvalues and eigenvectors for real symmetric matrices.

All prototypes for the Intel IPP matrix operating functions are placed in the “Syntax” sections of the corresponding chapters. The function flavors are grouped in the “Case” subsections in accordance with the type of operation.

The manual also includes an Index of major terms and definitions used in this volume.

Function Descriptions

In Chapters 3 through 8, each function is introduced by its short name (without the `ippm` prefix and descriptors) and a brief description of its purpose. This is followed by the full collection of prototypes for the specified function, definition of its parameters, and a more detailed explanation of the function purpose.The following sections are included in the function description:

Syntax	Contains all the prototypes of the specified function that are grouped in the “Case” subsections in accordance with the function flavor.
Parameters	Describes arguments for all flavors of the function.
Description	Defines the function and details the operation performed by the function. Code examples and equations that the function implements may be included in the description.
Return Values	Explains the value returned by the function. Commonly, it lists error codes that the function returns.

Audience for This Manual

This manual is intended for developers of applications that involve substantial use of operations on small matrices or may benefit from such operation. The audience must have experience using C and a working knowledge of the vocabulary and principles of matrix operations.

Notational Conventions

In this manual, notational conventions include:

- Fonts used for distinction between the text and the code
- Naming conventions for different items.

Font Conventions

The following font conventions are used throughout the manual:

`This type style`	Mixed with the uppercase in structure names as in `IppLibraryVersion`; also used in function names, code examples, and call statements; for example, `ippmAdd_mm_32f`.
`This type style`	Parameters in function type parameters and parameters description, for example, *`src1Stride1`*, *`width`*.

Naming Conventions

The following naming conventions for different items are used by the Intel IPP software:

- All names of the functions used for matrix operations have the `ippm` prefix. In code examples you can distinguish the Intel IPP interface functions from the application functions by this prefix.

NOTE. In this manual, the `ippm` prefix in function names is always used in code examples and function prototypes. In the text, this prefix is omitted when referring to the function group.

- Each new part of a function name starts with an uppercase character without underscore, for example, `ippmFrobNorm`. The underscore is used to separate the data types and data descriptors used with the function, for example, `ippmFrobNorm_ma_32f`.

For the detailed description of the function names structure in Intel IPP, see Function Naming in Chapter 2.

Getting Started 2

This chapter explains the purpose and structure of Intel® Integrated Performance Primitives (Intel® IPP) for small matrices.

The chapter also looks over the fundamental concepts used in the small matrix operations part of Intel IPP and defines function naming conventions used in the manual.

Purpose of Intel IPP for Small Matrices

Intel IPP for small matrices (IPP MX) actualizes linear algebra operations on matrices and vectors.

IPP MX provides solutions to a number of software development tasks that include development of various graphics, computer game applications and CAD applications. For example, you can find Intel IPP solutions useful for the applications that require transforming point coordinates from one coordinate system to another, computing dynamics for physical motion modeling, or solving systems of linear equations.

Data Types

Intel IPP for small matrices supports float point data with single and double precision.

Memory Layout

IPP MX supports vectors and matrices, whose elements are spaced in memory at equal intervals, as well as matrices and vectors, whose elements are arbitrary allocated in memory. This feature is unique to this library, more information on the types of matrices and vectors that the functions can operate on is given in IPP MX Objects and Object Description.

Arrays of Vectors and Matrices

Distinctive feature of IPP MX is matrix and vector arrays processing. For example, if there is an IPP MX function that can process two matrices, then there certainly exist analogous functions that process arrays of matrices, with one or both source operands being matrix arrays.

Matrix and vector arrays are processed element by element. Thus, the Add function for vector arrays adds the first vector of the first array to the first vector of the second array, then adds the second vectors of the two arrays, and so forth. The result is stored in the destination vector array. If the first operand is a single vector and the second is a vector array, then the function adds the single vector to each element of the array.

It is recommended to use these functions when processing large amounts of data, as it significantly accelerates your program.

Matrix Transposition

IPP MX library includes special functions that operate on transposed matrices and on arrays of transposed matrices. These functions were provided for such operations as Add, Sub, Mul and some others. For example, there are three IPP MX Add functions that add single matrices: the first function operates on two ordinary matrices, the second function operates on a transposed matrix and an ordinary matrix, and the third one operates on two transposed matrices.

When a function operates on a transposed matrix (or an array of transposed matrices), no special data is required. It is enough to specify non-transposed matrix (array of non-transposed matrices) as function operand, and the function will transpose this matrix (array of matrices) without performance loss.

In-Place Operations

IPP MX library does not contain any in-place operations. Thus, if the source and the destination addresses coincide or overlap, there may be discrepancies in the results shown by versions of Intel IPP optimized for different processors.

Optimization

IPP MX functions are optimized for operations on small matrices and small vectors, particularly for matrices of the size 3x3, 4x4, 5x5, 6x6, and for vectors of the length 3, 4, 5, 6.

Note that when operating on small arrays, for example, a matrix array made up of two or three matrices of the size 3x3, overhead caused by function call and input parameters' check may be greater than the optimization gain.

IPP MX Objects

IPP MX functions operate on the following objects:

- constant
- vector
- array of vectors
- matrix
- array of matrices
- transposed matrix
- array of transposed matrices

Constant

IPP MX constant is scalar value. Value type is Ipp32f or Ipp64f.

Vector

The simplest vector is a one-dimensional continuous array (see Figure 2-1, case A). In Intel IPP for small matrices, vectors have a more complicated structure. Any elements stored in memory can be combined into an IPP MX vector. In IPP MX there is a difference between regular and irregular vectors. The vector is called regular, if its elements are equally spaced in memory: see Figure 1-1, cases A, B. Otherwise, the vector is called irregular: see Figure 2-1, cases C, D.

Matrix

The simplest matrix is a two-dimensional continuous array (see Figure 2-2, case A). In Intel IPP for small matrices, matrices are represented by more complicated structures. Any elements stored in memory can be combined into an IPP MX matrix. In IPP MX there is a difference between

regular and irregular matrices. The matrix is called regular, if its row elements are equally spaced in memory and its rows are equally spaced in memory: see Figure 2-2, cases A, B. If one or both of these conditions is false, the matrix is called irregular: see Figure 2-2, cases C, D.

In this manual matrix sizes are given as *width* x *height*.

Figure 2-1 Regular and Irregular Vectors

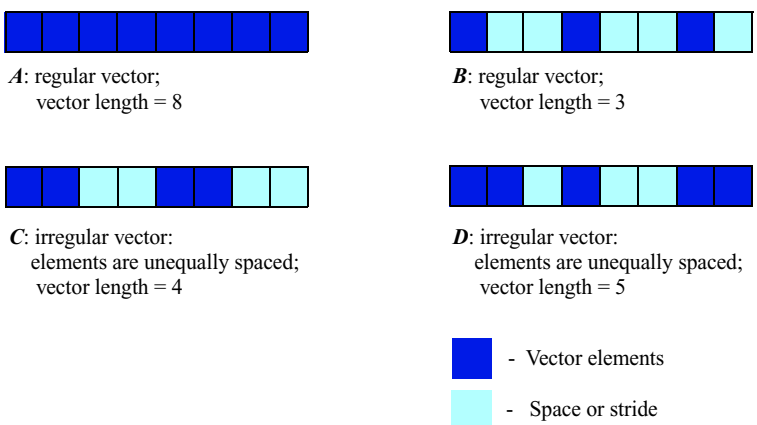
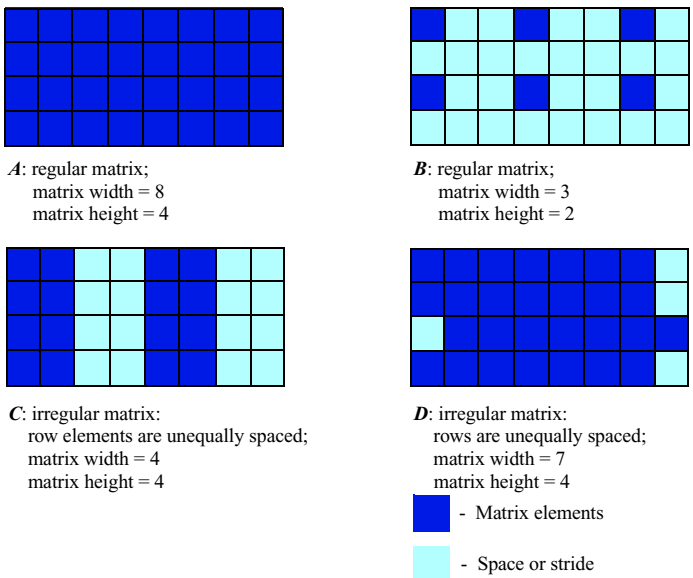


Figure 2-2 Regular and Irregular Matrices



Array of Vectors

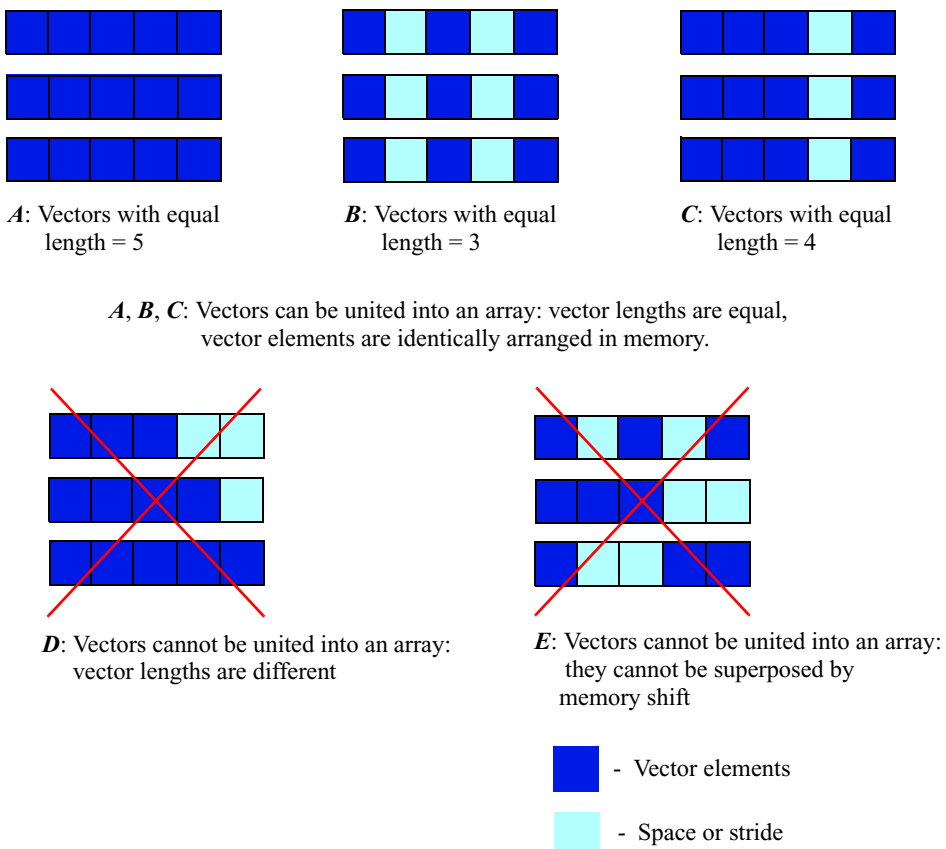
The way Intel IPP for small matrices operates on vector arrays is similar to its operation on single vectors. Thus, different vectors cannot be combined into one array, unless the following conditions are fulfilled:

vectors have equal length

vectors have identical structure, i.e. if superposed by memory shift, they will coincide.

See Figure 2-3, cases A, B, C for proper vector sets. If one or both of these conditions is not satisfied, vectors cannot be combined into an array (see Figure 2-3, cases D, E).

Figure 2-3 Combining Vectors into an Array



However, not all proper vectors can be combined into an array. Another important condition is vector layout.

The vector layout is called regular, if the vectors are equally spaced in memory; otherwise, the layout is called irregular. In these terms, vectors can be combined in the following way (see Figure 2-4, cases A, B, C):

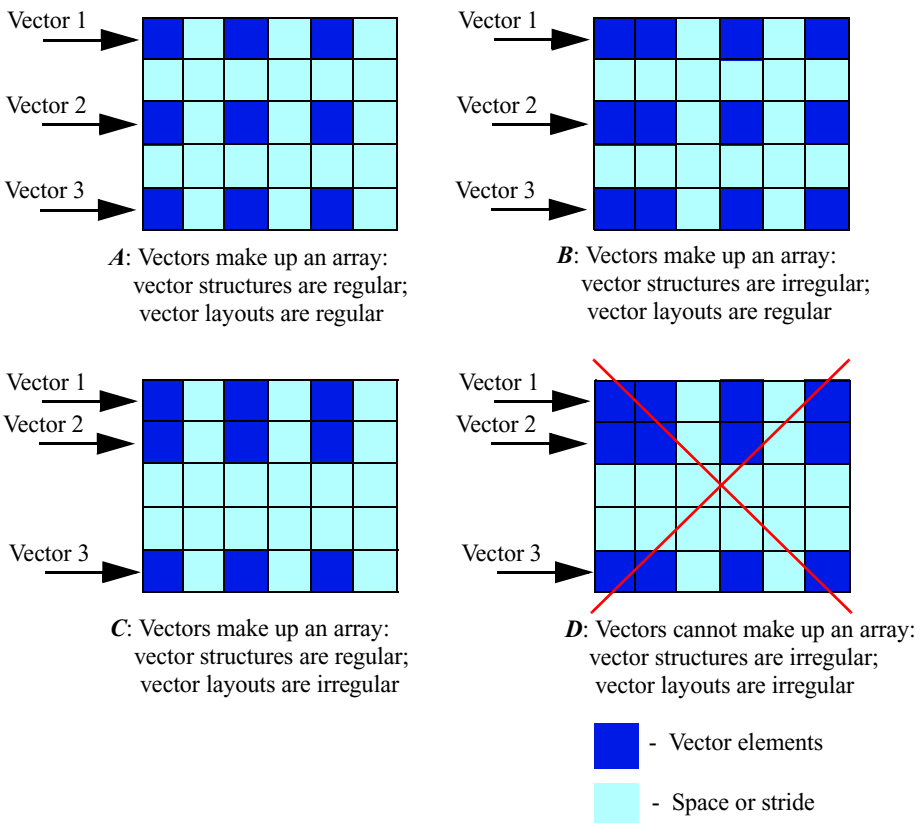
regular vectors with regular layout

regular vectors with irregular layout

irregular vectors with regular layout

Irregular vectors with irregular layout cannot be combined into an array (see Figure 2-4, case D).

Figure 2-4 Permissible Vector Arrays



Array of Matrices

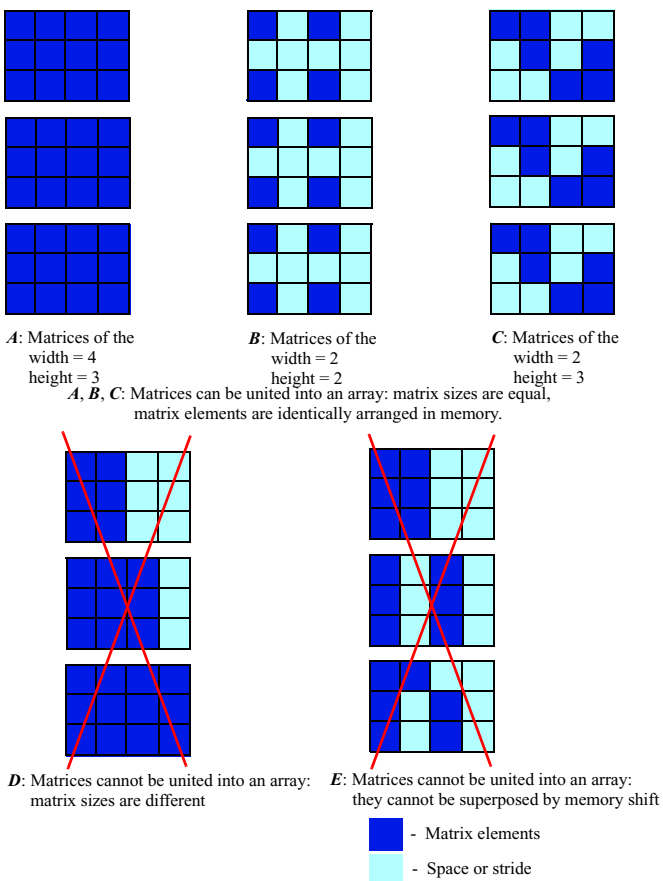
The way Intel IPP for small matrices operates on matrix arrays is similar to its operation on single matrices. Thus, different matrices cannot be combined into one array, unless the following conditions are fulfilled:

matrices have equal width and height

matrices have identical structure, i.e. if superposed by memory shift, they will coincide.

See Figure 2-5, cases A, B, C for proper matrix sets. If one or both of these conditions is not satisfied, matrices cannot be combined into an array (see Figure 2-5, cases D, E).

Figure 2-5 Combining Matrices into an Array



However, not all proper matrices can be combined into an array. Another important condition is matrix layout.

The matrix layout is called regular, if the matrices are equally spaced in memory; otherwise, the layout is called irregular. In these terms, matrices can be combined in the following way (see Figure 2-6, cases A, B, C):

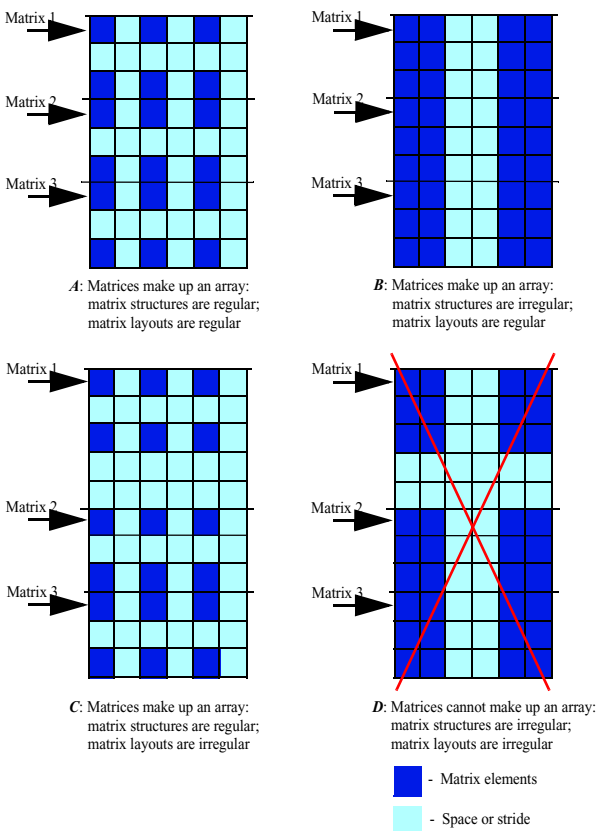
regular matrices with regular layout

regular matrices with irregular layout

irregular matrices with regular layout

Irregular matrices with irregular layout cannot be combined into an array (see Figure 2-6, case D).

Figure 2-6 Permissible Matrix Arrays



Transposed Matrix

When IPP MX functions operate on transposed matrices, these should be specified as function operands (see Matrix), and the function will transpose the necessary matrix or matrices during calculation.

Array of Transposed Matrices

When IPP MX functions operate on arrays of transposed matrices, the arrays should be specified as function operands (see Array of Matrices), and the function will transpose each matrix in the array during calculation.

Object Description

This section looks over several methods to specify objects that can be used as function arguments. The following terms should be defined before further presentation:

Object	A single matrix, single vector, array of matrices, or array of vectors that can serve as IPP MX function operand.
Object size	Object data that will be used by the IPP MX function for calculation, i.e. width and height of a matrix and length of a vector. If the operation is performed on a matrix or a vector array, object size is equal to the size of a single element in the array.

Description Methods

IPP MX library provides two methods of the description of a single matrix or a vector:

Standard description	The method is used when the matrix (vector) is regular (Figure 2-1 and Figure 2-2, cases A and B).
Pointer description	The method is used when the matrix (vector) is irregular (Figure 2-1 and Figure 2-2, cases C and D).

Note that the Standard description may be represented through the Pointer description, but not vice versa.

IPP MX library provides three methods of the description of matrix and vector arrays:

Standard description	The method is used when all the matrices (vectors) have regular structure. Matrices (vectors) must be regularly spaced in memory (Figure 2-4 and Figure 2-6, case A).
Pointer description	The method is used when all the matrices (vectors) have irregular structure. Matrices (vectors) must be regularly spaced in memory (Figure 2-4 and Figure 2-6, case B).

Layout description The method is used when all the matrices (vectors) have regular structure, but are irregularly spaced in memory (Figure 2-4 and Figure 2-6, case C).

NOTE. All elements in the array must have identical structure.

The following subsections describe the above methods in detail by means of IPP MX function parameters. There is no specification of object size, since although the size is an essential feature of an object, many IPP MX functions require only one size for several objects. The concept of size as a function and object attribute, as well as its specification is described in Object Size Puzzle.

Strides

If the data is regularly organized, it is easier to describe it in terms of strides. Intel IPP for matrix operations introduces three types of strides:

Stride 0	stride between matrices (vectors) in the array.
Stride 1	stride between matrix rows.
Stride 2	stride between vector elements or matrix row elements.

When operating on regular matrices, you should specify stride1 and stride2.

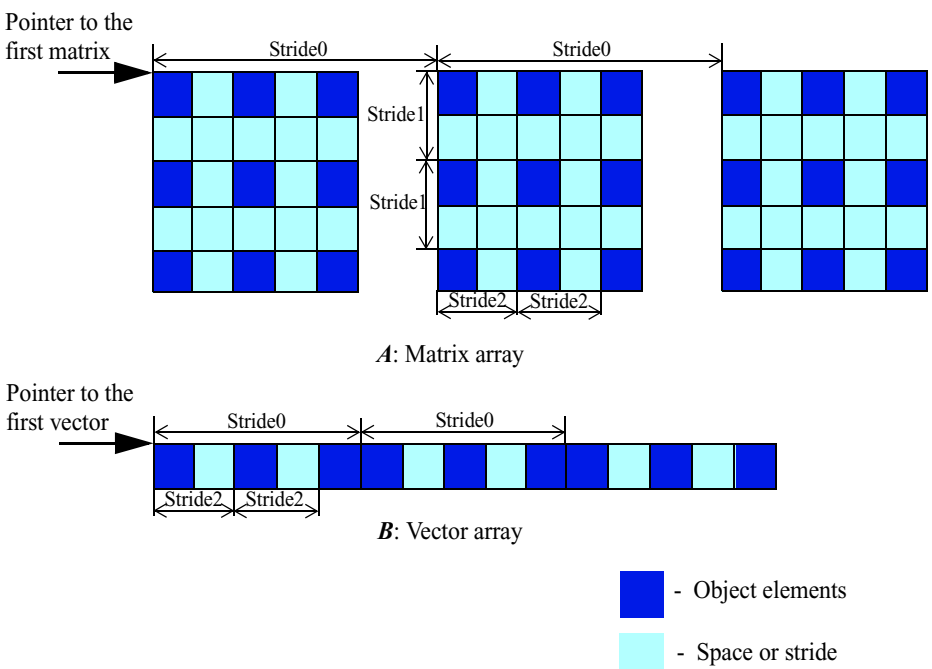
When operating on transposed matrices, you should never exchange stride1 and stride2. Instead, use the function that operates on transposed matrices.

NOTE. All strides are measured in bytes. Stride value must be positive and divisible by the size of the data type. To convert stride value measured in elements to the number of bytes you should multiply it by the size of the data type.

Standard Description

To describe an object by Standard method, specify one pointer to object's data and several strides. When the operation is performed on a single matrix or a vector, the required pointer is the pointer to the first object element. When the operation is performed on matrix or vector arrays, the required pointer is the pointer to the first element in the first matrix (vector) of the array.

Figure 2-7 Standard Description



The following fragment of C code describes the regular matrix array shown in Figure 2-7, A:

```
// Allocate memory for the array of continuous matrices
Ipp32f pMatrices[5*5*3];
// Set stride2
int stride2 = sizeof(Ipp32f)*2;
// Set stride1
```

```
int stride1 = sizeof(Ipp32f)*10;
// Set stride0
int stride0 = sizeof(Ipp32f)*25;
   * * *
// Call IPP MX function
ippm<Operation>_ma_32f(…, pMatrices, stride0, stride1, stride2, …);
```

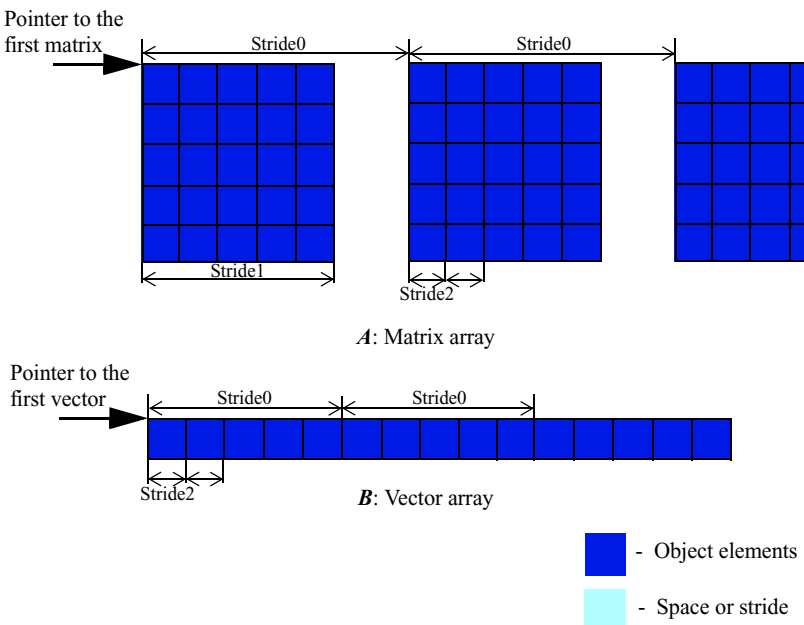
The next code fragment represents description for regular array of vectors shown in Figure 2-7, B:

```
// Allocate memory for the array of continuous vectors
Ipp32f pVectors[5*3];
// Set stride2
int stride2 = sizeof(Ipp32f)*2;
// Set stride0
int stride0 = sizeof(Ipp32f)*5;
   * * *
// Call IPP MX function
ippm<Operation>_va_32f(…,pVectors, stride0, stride2, …);
```

Single matrices and vectors are described in the same way without stride 0 specification.

NOTE. Strides are calculated in bytes.

Figure 2-8 Standard Description (Continuous Object)



Continuous objects also belong to the general regular case. Thus, all strides including stride 2 must be specified. When operating on the matrix array shown in Figure 2-8, case A, specify the following strides (strides are calculated for Ipp32f data type):

```
int stride2 = sizeof(Ipp32f);
int stride1 = stride2*width;
int stride0 = stride1*height;
```

When operating on the vector array shown in Figure 2-8, case B, specify the following strides

```
int stride2 = sizeof(Ipp32f);
int stride0 = stride2*length;
```

Pointer Description

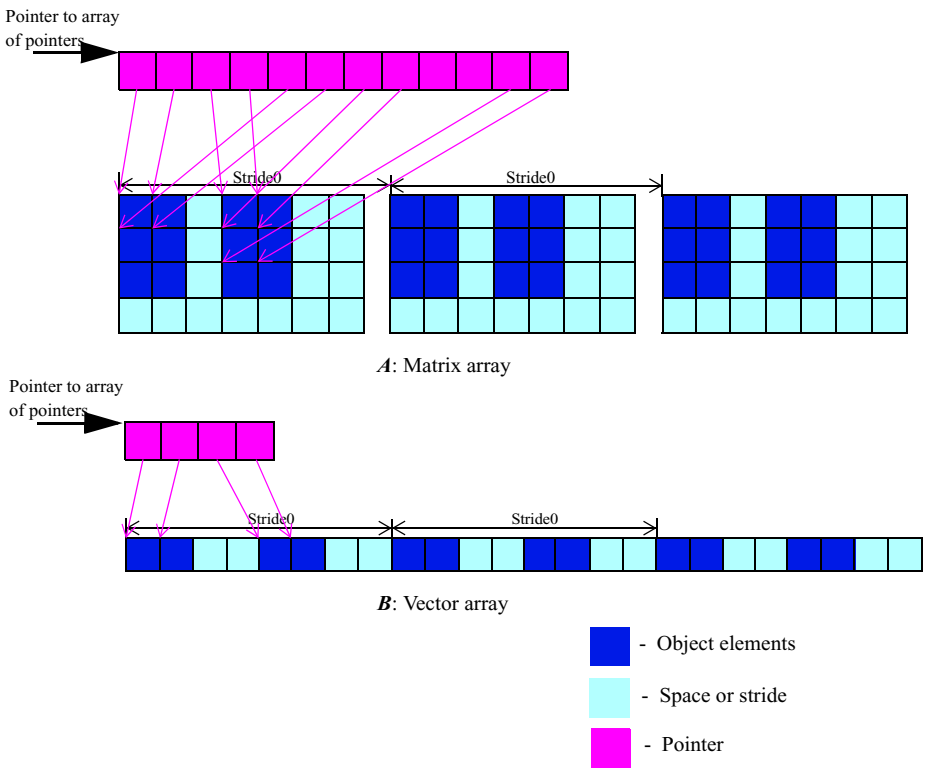
Pointer method is used when you deal with objects of irregular structure. A convenient way to describe an irregular structure is creation of a mask for the data. The Pointer method creates masks for objects by direct pointing to every element of a vector (matrix). Pointers to the elements make up an array.

To describe an object by the Pointer method, specify the pointer to the array of pointers and roiShift in bytes (The concept of the roiShift parameter will be clarified later; in this discussion it is set to 0.). If the operation is performed on matrix or vector arrays, also specify the stride between the matrices (vectors), i.e. stride 0.

To apply the Pointer description method, first, create an array of pointers with the length equal to the number of elements in a single matrix (vector). Then initialize this array by making its elements point to every element of the single matrix (vector). When processing matrix or vector arrays, the pointers must point to the elements of the first matrix (vector).

IPP MX function that operates on objects described by the Pointer method has suffix `_P`. If a function has the `_P` suffix, then all objects, except constants, must be defined by Pointer method.

Figure 2-9 Pointer Description



The following fragment of C code describes the matrix array shown in Figure 2-9, case A:

```
// Allocate memory for the array of continuous matrices
Ipp32f  pMatrices[8*4*3];
// Assign pointer to the first continuous matrix
Ipp32f* pFirstMatrix = pMatrices;
// Allocate special array for pointers to matrix elements
Ipp32f* ppPointer[4*3];
// Declare subsidiary descriptors
int roiShift;
int stride0;
```

```
   * * *
// Set first row of matrix
ppPointer [0] = pFirstMatrix + 0;
ppPointer [1] = pFirstMatrix + 1;
ppPointer [2] = pFirstMatrix + 4;
ppPointer [3] = pFirstMatrix + 5;
// Set second row of matrix
ppPointer [4] = pFirstMatrix + 8 + 0;
ppPointer [5] = pFirstMatrix + 8 + 1;
ppPointer [6] = pFirstMatrix + 8 + 4;
ppPointer [7] = pFirstMatrix + 8 + 5;
// Set third row of matrix
ppPointer [ 8] = pFirstMatrix + 8*2 + 0;
ppPointer [ 9] = pFirstMatrix + 8*2 + 1;
ppPointer [10] = pFirstMatrix + 8*2 + 4;
ppPointer [11] = pFirstMatrix + 8*2 + 5;
// Set roiShift
roiShift = 0;
// Set stride0
stride0 = sizeof(Ipp32f)*8*4;
// Call IPP MX function
ippm<Operation>_ma_32f_P(…, ppPointer, roiShift, stride0, …);
```

The following fragment of C code describes the vector array shown in Figure 2-9, case B:

```
// Allocate memory for the array of continuous vectors
Ipp32f  pVectors[8*3];
// Assign pointer to the first continuous vector
Ipp32f* pFirstVector = pVectors;
// Allocate special array for pointers to vector elements
Ipp32f* ppPointer[4];
// Declare subsidiary descriptors
int roiShift;
int stride0;
```

```
   * * *
// Set pointers
ppPointer [0] = pFirstVector + 0;
ppPointer [1] = pFirstVector + 1;
ppPointer [2] = pFirstVector + 4;
ppPointer [3] = pFirstVector + 5;
// Set roiShift
roiShift = 0;
// Set stride0
stride0 = sizeof(Ipp32f)*8;
// Call IPP MX function
ippm<Operation>_va_32f_P(…, ppPointer, roiShift, stride0, …);
```

Single matrices and vectors are described in the same way without stride 0 specification.

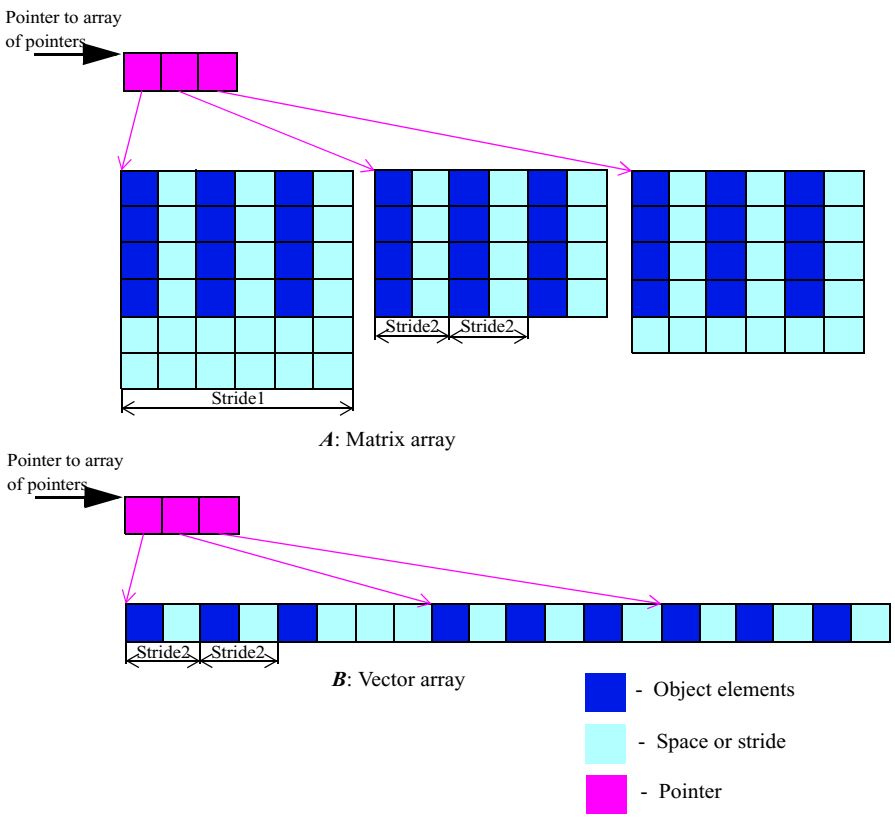
Layout Description

Layout description method is used when dealing with matrix or vector arrays. Unlike the Pointer description method, which defines matrix elements by pointers and matrix layout by strides, the Layout method defines matrix elements by strides and matrix layout by pointers.

To describe an object by the Layout method, specify pointers to each matrix or vector in the array, roiShift (in bytes), stride 2, and stride 1 (The concept of the roiShift parameter will be clarified later; in this discussion it is set to 0.). Note that no stride 0 specification is required. Create a special array of pointers with the length equal to the number of array components. Initialize the array by making its elements point to every matrix (vector). Specify the strides required to define single matrix (vector).

IPP MX function that operates on objects described by the Layout method has suffix `_L`. If a function has the `_L` suffix, then matrix and vector arrays must be defined by the Layout method and single matrices or vectors by the Standard method.

Figure 2-10 Layout Description



The following fragment of C code describes the matrix array shown in Figure 2-10, case A:

```
// Allocate memory for three continuous matrices
Ipp32f pFirstMatrix [6*5];
Ipp32f pSecondMatrix[6*5];
Ipp32f pThirdMatrix [6*5];
// Allocate special array for pointers to 3 matrices:
Ipp32f* ppLayout[3];
// Declare subsidiary descriptors
```

```
int roiShift;
int stride1, stride2;
   * * *
// Set pointers to matrices
ppLayout [0] = pFirstMatrix;
ppLayout [1] = pSecondMatrix;
ppLayout [2] = pThirdMatrix;
// Set roiShift
roiShift = 0;
// Set stride2
stride2 = sizeof(Ipp32f)*2;
// Set stride1
stride1 = sizeof(Ipp32f)*6;
// Call IPP MX function
ippm<Operation>_ma_32f_L(…, ppLayout, roiShift, stride1, stride2, …);
```

The following fragment of C code describes the vector array shown in Figure 2-10, case B:

```
// Allocate memory for three continuous vectors
Ipp32f pFirstVector [6];
Ipp32f pSecondVector[6];
Ipp32f pThirdVector [6];
// Allocate special array for pointers to 3 vectors
Ipp32f* ppLayout[3];
// Declare subsidiary descriptors
int roiShift;
int stride2;
   * * *
// Set pointers to vectors
ppLayout [0] = pFirstVector;
ppLayout [1] = pSecondVector;
ppLayout [2] = pThirdVector;
// Set roiShift
roiShift = 0;
```

```
// Set stride2
stride2 = sizeof(Ipp32f)*2;
// Call IPP MX function
ippm<Operation>_va_32f_L(…, ppLayout, roiShift, stride2, …);
```

RoiShift Parameter

To illustrate the concept of the roiShift (region of interest shift) parameter, have another look at Figure 2-10. In the described objects, odd columns of continuous matrices and odd elements of continuous vectors are processed. However, to process even columns or elements, an additional parameter may be needed to specify the shift (in this case, one-element shift right).

If the Standard description method is used, to describe data shifted this way, it is sufficient to shift only one data pointer. However, with the Layout method used, it is necessary to shift every pointer in the layout array by the same number of elements (in this case, one), or bytes. Similar problem arises when the Pointer description is used: to shift the mask, all elements in the pointer array must be shifted accordingly. This shift is specified using the roiShift parameter, which is required for Pointer and Layout description methods. RoiShift specifies the number of bytes by which IPP MX function will shift each pointer in the pointer array. If no shifting is needed (as in Figure 2-9 and Figure 2-10), roiShift is set to 0.

NOTE. RoiShift parameter is measured in bytes. RoiShift value can be equal to 0. Otherwise, roiShift value must be positive and divisible by the size of the data type. To convert roiShift value measured in elements to the number of bytes you should just multiply it by the size of the data type.

Figure 2-11 Pointer Description with RoiShift

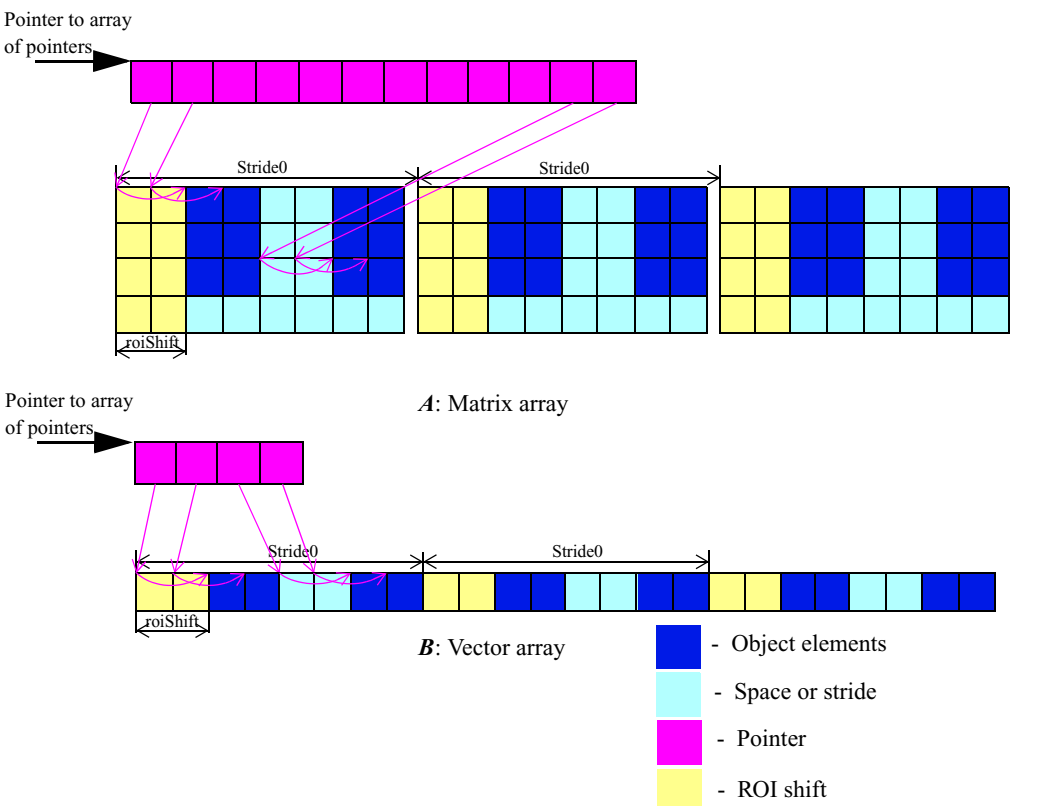


Figure 2-11, case A shows the matrix array that is a result of Layout description with non-zero roiShift parameter.

Figure 2-11, case B shows the vector array that is a result of Layout description with non-zero roiShift parameter.

The following fragment of C code specifies roiShift for both A and B cases:

```
// Set roiShift
roiShift = sizeof(32f);
```

Object Descriptors Table

The following table contains subsidiary object descriptors required for each particular case. All description methods and all object types are collected here. The order of descriptors in the table is the order in which IPP MX function parameters are presented.

Table 2-1 Object Descriptors

Description	Object	roiShift	stride0	stride1	stride2
Standard	Matrix	-	-	+	+
	Array of matrices	-	+	+	+
	Vector	-	-	-	+
	Array of vectors	-	+	-	+
Pointer	Matrix	+	-	-	-
	Array of matrices	+	+	-	-
	Vector	+	-	-	-
	Array of vectors	+	+	-	-
Layout	Array of matrices	+	-	+	+
	Array of vectors	+	-	-	+

Function Naming

The function names in the small matrix operations domain of Intel IPP begin with the ippm prefix and have the following general format:

```
ippm<name>_<objects>_<datatype>[_descriptor](<arguments>);
```

Name

The `<name>` is an abbreviation for the core function operation, for example, “Add”, “Copy”. The `<name>` may consist of several functional parts. Each new part of a function name starts with an uppercase character, without underscore, for example, `ippmFrobNorm`.

Objects

`objects = <objecttype1>[<objecttype2>]`

Object type describes the type of source objects passed to a function for processing and may be the following:

`c`	constant
`v`	vector
`va`	array of vectors
`m`	matrix
`t`	transposed matrix
`ma`	array of matrices
`ta`	array of transposed matrices.

If both source objects have the same type, double type is defined. The `<objecttype2>` is omitted if the function has only one source object.

The type of destination object is declared, if the function has no source objects. Otherwise, the destination type is determined by source objects' types and by function operation.

Data Types

The current version of Intel IPP supports the following data types of the source and destination for functions that perform matrix operations:

`32f`	32-bit floating-point
`64f`	64-bit double precision.

All objects of each particular IPP MX function must have the same type. Accordingly, it is assumed that sizes of the objects are specified in sizes of the data type, unless otherwise explicitly indicated.

Descriptor

The `<descriptor>` field defines object description method (see Object Description) and contains the following symbols:

`S` Standard description.

`P` Pointer description.

`L` Layout description.

The default for Intel IPP matrix operating functions is the Standard description method.

Practically all IPP MX function names have one descriptor symbol or no descriptor at all, which allows them to be specified by Standard object description. The only exception is Copy functions that can copy data from one description to another. Copy function name contains two descriptors to define the source and the destination description methods.

If IPP MX function name has the L descriptor, while the function itself operates not only on matrix and vector arrays, but also on single matrices (vectors), then the single matrix or vector must be described by Standard method, since the Layout method does not work on single objects.

Arguments

The `<arguments>` field specifies the function parameters. The order of arguments is as follows:

- all source objects (constants follow matrices and vectors)
- all destination objects
- count – number of matrices (vectors) in arrays (if the function operates on matrix or vector arrays)
- other, operation-specific operands.

The order of arguments specifying non-constant object is as follows:

- pointer to object data
- subsidiary object descriptors (see Object Descriptors Table)
- object size – optional arguments.

Object size is not obligatory for all objects. Object size arguments may be as follows:

- *`width`*, *`height`* - matrix width and height
- *`widthHeight`* - square matrix width and height
- *`length`* - vector length.

Parameter name convention

The parameter name has the following conventions:

- All parameters defined as pointers to any object start with p, for example, *pSrc*, *pDst*; all parameters defined as double pointers (pointers to the pointers) start with *pp*, for example, *ppSrc*, *ppDst*.
- All parameters defined as values start with a lowercase letter, for example, *val*, *len*, *count*.
- Each new part of a parameter name starts with an uppercase letter, without underscore, for example, *pSrc*, *srcStride2*.
- Each parameter name specifies its functionality. Source parameters named *pSrc* or *src* are sometimes followed by names or numbers, for example, *pSrc2*, *src2Len*.
- Output parameters named pDst or dst are followed by names or numbers, for example, *pDst*, *dstLen*.

Object Size Puzzle

As it was said in the Function Naming subsection (see Arguments), object size is not always required when describing an object. The majority of IPP MX functions specify several objects and only one object size. There exist certain rules that define the object size parameter in such cases:

- If the object is followed by object size, this is the size of the object.
- If there is no object size, this parameter is calculated by another object's size based on the function purpose and object types.

Example 1:

```
IppStatus ippmTranspose_m_32f(const Ipp32f* pSrc, int srcStride1, int
   srcStride2, int width, int height, Ipp32f* pDst, int dstStride1, int
   dstStride2);
```

Object size *width*, *height* follows the *src* matrix, therefore, it is the *src* size

srcWidth = width, srcHeight = height

The function purpose is transposition, therefore

dstWidth = height, dstHeight = width

Example 2:

```
IppStatus ippmAdd_mm_32f(const Ipp32f* pSrc1, int src1Stride1, int
   src1Stride2, const Ipp32f* pSrc2, int src2Stride1, int src2Stride2,
   Ipp32f* pDst, int dstStride1, int dstStride2, int width, int height);
```

Object size *width*, *height* follows the *dst* matrix, therefore, it is the *dst* size

dstWidth = width, dstHeight = height

The function purpose is addition of matrices, therefore the sizes of the elements should be equal to the size of *dst*.

src1Width = width, src1Height = height

src2Width = width, src2Height = height

Example 3:

```
IppStatus ippmAdd_tm_32f(const Ipp32f* pSrc1, int src1Stride1, int
   src1Stride2, const Ipp32f* pSrc2, int src2Stride1, int src2Stride2,
   Ipp32f* pDst, int dstStride1, int dstStride2, int width, int height);
```

Object size *width*, *height* follows the *dst* matrix, therefore, it is the *dst* size

dstWidth = width, dstHeight = height

The function purpose is addition of matrices, therefore the sizes of the elements should be equal to the size of *dst*.

item1Width = width, item1Height = height

item2Width = width, item2Height = height

The first object type is a transposed matrix. Therefore, *src1* stored in the memory has the following size

src1Width = item1Height = height,

src1Height = item1Width = width

and after the transposition, you will get the right first object size.

The second object type is a single matrix, therefore

src2Width = item2Width = width,

src2Height = item2Height = height

Example 4:

```
IppStatus ippmMul_mm_32f(const Ipp32f* pSrc1, int src1Stride1, int
   src1Stride2, int matr1Width, int matr1Height, const Ipp32f* pSrc2,
   int src2Stride1, int src2Stride2, int matr2Width, int matr2Height,
   Ipp32f* pDst, int dstStride1, int dst2Stride2)
```

Object sizes follow each of the *src* matrices, therefore, these are the *src* sizes

src1Width = matr1Width, src1Height = matr1Height

src2Width = matr2Width, src2Height = matr2Height

The function purpose is multiplication of matrices, therefore the width of the product is equal to the width of the second multiplier, and the height of the product is equal to the height of the first multiplier. The *dst* sizes must be

dstWidth = matr2Width, dstHeight = matr1Height

Example 5:

```
IppStatus ippmMul_tm_32f(const Ipp32f* pSrc1, int src1Stride1, int
   src1Stride2, int matr1Width, int matr1Height, const Ipp32f* pSrc2,
   int src2Stride1, int src2Stride2, int matr2Width, int matr2Height,
   Ipp32f* pDst, int dstStride1, int dst2Stride2)
```

Object sizes follow each of the *src* matrices, therefore, these are the *src* sizes

src1Width = matr1Width, src1Height = matr1Height

src2Width = matr2Width, src2Height = matr2Height

The function purpose is multiplication of matrices, therefore the width of the product is equal to the width of the second multiplier, and the height of the product is equal to the height of the first multiplier.

The first object type is a transposed matrix, while the second object type is an ordinary matrix. Therefore, the sizes the multipliers are equal.

efficient1Width = matr1Height, efficient1Height = matr1Width

efficient2Width = matr2Width, efficient2Height = matr2Height

The *dst* sizes must be

dstWidth = efficient2Width = matr2Width,

dstHeight = efficient1Height = matr1Width

Error Reporting

The Intel IPP functions return the status of the performed operation to report errors and warnings to the calling program. Thus, it is up to the application to perform error-related actions and/or recover from the error. The last value of the error status is not stored, and the user is to decide whether to check it or not as the function returns. The status values are of IppStatus type and are global constant integers.

Below you can see a list of status codes and corresponding messages reported by the Intel IPP for small matrices.

`ippStsSizeMatchMatrixErr`	Unsuitable sizes of the source matrices.
`ippStsCountMatrixErr`	Count parameter is negative or equal to 0.
`ippStsRoiShiftMatrixErr`	RoiShift is negative or not divisible by the size of the data type.
`ippStsStrideMatrixErr`	Stride value is not positive or not divisible by the size of the data type.
`ippStsSingularErr`	Matrix is singular.
`ippStsNotPosDefErr`	Not positive-definite matrix.
`ippStsSizeErr`	Wrong value of the data size.
`ippStsNoErr`	No error, it's OK.
`ippStsDivByZeroErr`	An attempt to divide by zero.
`ippStsNullPtrErr`	Null pointer error.
`ippStsConvergeErr`	Indicates an error if the algorithm does not converge.

The status codes ending with `Err` (except for the `ippStsNoErr` status) indicate an error; the integer values of these codes are negative. When an error occurs, the function execution is interrupted.

For example, if the source matrix for `ippmLUDecomp` is singular, the function stops execution and returns with the error status `ppStsSingularErr`. If the input stride value is 0 or 3, the function stops execution and returns with the error status `ippStsStrideMatrixErr`.

Code Examples

The manual contains a number of code examples to demonstrate both some particular features of the small matrix functions and how these functions can be called.

Many of these code examples output result data together with status code and associated messages in case when error condition was met.

To keep the example code simpler, special definitions of print statements are used for better representation of results, as well as print status codes and messages.

The code definitions given below make it possible to build the examples contained in the manual by straightforward copying and pasting the example code fragments.

Example 2-1 Code Definitions

```
/*
// The functions providing simple output of the result
// for single precision and double precision real data.
// These functions are only for tight data:
//   Stride2 = sizeof(dataType)
//   Srtide1 = width*sizeof(dataType)
//   Stride0 = length*sizeof(dataType) - for vector array
//   Stride0 = width*height*sizeof(dataType) - for matrix array
*/

#define genPRINT_m(TYPE) \
void printf_m_Ipp##TYPE(const char* msg, Ipp##TYPE* buf, \
                        int width, int height, IppStatus st ) \
{   int i, j; \
    if( st < ippStsNoErr ) { \
        printf( "-- error %d, %s\n", st, ippGetStatusString( st )); \
    } else { \
        printf("%s \n", msg ); \
        for( j=0; j <  height; j++) { \
                 for( i=0; i < width; i++) { \
                   printf("%f  ", buf[j*width+i]); } \
                 printf("\n"); } } \
}
```

Example 2-1 Code Definitions (continued)

```
#define genPRINT_ma(TYPE) \
void printf_ma_Ipp##TYPE(const char* msg, Ipp##TYPE## *buf, \
                      int width, int height, int count, IppStatus st ) \
{   int i, j, k; \
    if( st < ippStsNoErr ) { \
        printf( "-- error %d, %s\n", st, ippGetStatusString( st )); \
    } else { \
        printf("%s \n", msg ); \
        for( j=0; j <  height; j++) { \
                for( k=0; k < count; k++) { \
                for( i=0; i < width; i++){ \
                   printf("%f ", buf[j*width+i+k*width*height]); \
                } printf("    "); } printf("\n");}} \
}

#define genPRINT_m_L(TYPE) \
void printf_ma_Ipp##TYPE##_L(const char* msg, Ipp##TYPE** buf, \
                      int width, int height, int count, IppStatus st )\
{   int i, j, k; \
    Ipp##TYPE## *dst;  \
    if( st < ippStsNoErr ) { \
        printf( "-- error %d, %s\n", st, ippGetStatusString( st )); \
    } else { \
        printf("%s \n", msg ); \
        for( j=0; j <  height; j++) { \
                for( k=0; k < count; k++) { \
                    dst = (Ipp##TYPE##*)buf[k]; \
                for( i=0; i < width; i++) { \
                      printf("%f ", dst[j*width+i]); } \
                printf("    "); } printf("\n"); } } \
}
```

Example 2-1 Code Definitions (continued)

```
#define genPRINT_m_P(TYPE) \
void printf_m_Ipp##TYPE##_P(const char* msg, Ipp##TYPE** buf, \
                        int width, int height, IppStatus st ) \
{   int i, j; \
    if( st < ippStsNoErr ) { \
        printf( "-- error %d, %s\n", st, ippGetStatusString( st )); \
    } else { \
        printf("%s \n", msg ); \
        for( j=0; j <  height; j++) { \
            for( i=0; i < width; i++) { \
                  printf("%f ", *buf[j*width+i]); } \
                printf("\n"); } } \
}

#define genPRINT_va(TYPE) \
void printf_va_Ipp##TYPE(const char* msg, Ipp##TYPE* buf, \
                      int length, int count, IppStatus st ) \
{   int i, j; \
    if( st < ippStsNoErr ) { \
        printf( "-- error %d, %s\n", st, ippGetStatusString( st )); \
    } else { \
        printf("%s \n", msg ); \
        for( j=0; j <  count; j++) { \
                for( i=0; i < length; i++) { \
                  printf("%f  ", buf[j*length+i]); } \
                printf("\n"); } } \
}
```

Example 2-1 Code Definitions (continued)

```
void printf_v_int(const char* msg, int* buf, int length) \
{   int i; \
    printf("%s \n", msg ); \
    for( i=0; i < length; i++) \
          printf("%d  ", buf[i]); \
    printf("\n"); \
}

genPRINT_va( 32f );
genPRINT_m( 32f );
genPRINT_ma( 32f );
genPRINT_m_P( 32f );
genPRINT_m_L( 32f );

genPRINT_va( 64f );
genPRINT_m( 64f );
genPRINT_ma( 64f );
genPRINT_m_P( 64f );
genPRINT_m_L( 64f );
```

Utility Functions 3

This chapter describes Intel® Integrated Performance Primitives (Intel® IPP) functions for small matrices that copy an object of any type to another object of any type, extract Regions of Interest (ROI), and initialize matrices.

Table 3-1 Utility functions

Function Base Name	Operation
Copy	Performs copy operation.
Extract	Performs ROI extraction.
LoadIdentity	Initializes identity matrix.

Copy

Performs copy operation.

Syntax

Case 1: Vector array operation

```
IppStatus ippmCopy_va_32f_SS(const Ipp32f* pSrc, int srcStride0,
    int srcStride2, Ipp32f* pDst, int dstStride0, int dstStride2, int len,
    int count);
IppStatus ippmCopy_va_64f_SS(const Ipp64f* pSrc, int srcStride0,
    int srcStride2, Ipp64f* pDst, int dstStride0, int dstStride2, int len,
    int count);
```

```
IppStatus ippmCopy_va_32f_SP(const Ipp32f* pSrc, int srcStride0,
   int srcStride2, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int len,
   int count);
IppStatus ippmCopy_va_64f_SP(const Ipp64f* pSrc, int srcStride0,
   int srcStride2, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int len,
   int count);
IppStatus ippmCopy_va_32f_SL(const Ipp32f* pSrc, int srcStride0,
   int srcStride2, Ipp32f** ppDst, int dstRoiShift, int dstStride2, int len,
   int count);
IppStatus ippmCopy_va_64f_SL(const Ipp64f* pSrc, int srcStride0,
   int srcStride2, Ipp64f** ppDst, int dstRoiShift, int dstStride2, int len,
   int count);
IppStatus ippmCopy_va_32f_LS(const Ipp32f** ppSrc, int srcRoiShift,
   int srcStride2, Ipp32f* pDst, int dstStride0, int dstStride2, int len,
   int count);
IppStatus ippmCopy_va_64f_LS(const Ipp64f** ppSrc, int srcRoiShift,
   int srcStride2, Ipp64f* pDst, int dstStride0, int dstStride2, int len,
   int count);
IppStatus ippmCopy_va_32f_PS(const Ipp32f** ppSrc, int srcRoiShift,
   int srcStride0, Ipp32f* pDst, int dstStride0, int dstStride2, int len,
   int count);
IppStatus ippmCopy_va_64f_PS(const Ipp64f** ppSrc, int srcRoiShift,
   int srcStride0, Ipp64f* pDst, int dstStride0, int dstStride2, int len,
   int count);
IppStatus ippmCopy_va_32f_LP(const Ipp32f** ppSrc, int srcRoiShift,
   int srcStride2, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int len,
   int count);
IppStatus ippmCopy_va_64f_LP(const Ipp64f** ppSrc, int srcRoiShift,
   int srcStride2, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int len,
   int count);
IppStatus ippmCopy_va_32f_LL(const Ipp32f** ppSrc, int srcRoiShift,
   int srcStride2, Ipp32f** ppDst, int dstRoiShift, int dstStride2, int len,
   int count);
IppStatus ippmCopy_va_64f_LL(const Ipp64f** ppSrc, int srcRoiShift,
   int srcStride2, Ipp64f** ppDst, int dstRoiShift, int dstStride2, int len,
   int count);
IppStatus ippmCopy_va_32f_PP(const Ipp32f** ppSrc, int srcRoiShift,
   int srcStride0, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int len,
   int count);
```

```
IppStatus ippmCopy_va_64f_PP(const Ipp64f** ppSrc, int srcRoiShift,
   int srcStride0, Ipp64f** pDst, int dstRoiShift, int dstStride0, int len,
   int count);

IppStatus ippmCopy_va_32f_PL(const Ipp32f** ppSrc, int srcRoiShift,
   int srcStride0, Ipp32f** ppDst, int dstRoiShift, int dstStride2, int len,
   int count);

IppStatus ippmCopy_va_64f_PL(const Ipp64f** ppSrc, int srcRoiShift,
   int srcStride0, Ipp64f** ppDst, int dstRoiShift, int dstStride2, int len,
   int count);
```

Case 2: Matrix array operation

```
IppStatus ippmCopy_ma_32f_SS(const Ipp32f* pSrc, int srcStride0,
   int srcStride1, int srcStride2, Ipp32f* pDst, int dstStride0,
   int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmCopy_ma_64f_SS(const Ipp64f* pSrc, int srcStride0,
   int srcStride1, int srcStride2, Ipp64f* pDst, int dstStride0,
   int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmCopy_ma_32f_SP(const Ipp32f* pSrc, int srcStride0,
   int srcStride1, int srcStride2, Ipp32f** ppDst, int dstRoiShift,
   int dstStride0, int width, int height, int count);

IppStatus ippmCopy_ma_64f_SP(const Ipp64f* pSrc, int srcStride0,
   int srcStride1, int srcStride2, Ipp64f** ppDst, int dstRoiShift,
   int dstStride0, int width, int height, int count);

IppStatus ippmCopy_ma_32f_SL(const Ipp32f* pSrc, int srcStride0,
   int srcStride1, int srcStride2, Ipp32f** ppDst, int dstRoiShift,
   int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmCopy_ma_64f_SL(const Ipp64f* pSrc, int srcStride0,
   int srcStride1, int srcStride2, Ipp64f** ppDst, int dstRoiShift,
   int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmCopy_ma_32f_LS(const Ipp32f** ppSrc, int srcRoiShift,
   int srcStride1, int srcStride2, Ipp32f* pDst, int dstStride0,
   int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmCopy_ma_64f_LS(const Ipp64f** ppSrc, int srcRoiShift,
   int srcStride1, int srcStride2, Ipp64f* pDst, int dstStride0,
   int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmCopy_ma_32f_PS(const Ipp32f** ppSrc, int srcRoiShift,
   int srcStride0, Ipp32f* pDst, int dstStride0, int dstStride1,
   int dstStride2, int width, int height, int count);
```

```
IppStatus ippmCopy_ma_64f_PS(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);
IppStatus ippmCopy_ma_32f_LP(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, Ipp32f** ppDst, int dstRoiShift,
    int dstStride0, int width, int height, int count);
IppStatus ippmCopy_ma_64f_LP(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, Ipp64f** ppDst, int dstRoiShift,
    int dstStride0, int width, int height, int count);
IppStatus ippmCopy_ma_32f_LL(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, Ipp32f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int width, int height, int count);
IppStatus ippmCopy_ma_64f_LL(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, Ipp64f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int width, int height, int count);
IppStatus ippmCopy_ma_32f_PP(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int width,
    int height, int count);
IppStatus ippmCopy_ma_64f_PP(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int width,
    int height, int count);
IppStatus ippmCopy_ma_32f_PL(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int width, int height, int count);
IppStatus ippmCopy_ma_64f_PL(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int width, int height, int count);
```

Parameters

pSrc, ppSrc	Pointer to the source object or array of objects.
srcStride0	Stride between the objects in the source array.
srcStride1	Stride between the rows in the source matrix(ces).
srcStride2	Stride between the elements in the source object.
srcRoiShift	ROI shift in the source object.
pDst, ppDst	Pointer to the destination object or array of objects.
dstStride0	Stride between the objects in the destination array.

dstStride1	Stride between the rows in the destination matrix.
dstStride2	Stride between the elements in the destination object.
dstRoiShift	ROI shift in the destination object.
width	Matrix width.
height	Matrix height.
len	Vector length.
count	Number of objects in the array.

Description

The function `ippmCopy` is declared in the `ippm.h` header file. The function copies an object of any type to another object of any type and stores the result in the destination object.

If performed on matrices, all matrices involved in the operation must have the number of columns not less than *width* and the number of rows not less than *height*.

The following example demonstrates how to use the function `ippmCopy_va_32f_PS`. For more information, see also examples in the Getting Started chapter.

Example 3-1 ippmCopy_va_32f_PS

```
IppStatus copy_va_32f_PS(void) {
    /* Source data:  */
    Ipp32f a[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    /*
    // Pointer description for source data of interest a[0], a[6], a[7]:
    */
    Ipp32f* ppSrc[3] = { a, a+6, a+7 }; /* pointers array */
    int srcRoiShift = 0;
   int srcStride0  = sizeof(Ipp32f); /* formally must be initialized */
```

Example 3-1 ippmCopy_va_32f_PS (continued)

```
    /*
    // Standard description for destination vector
    */
    Ipp32f pDst[3];
    int dstStride2 = sizeof(Ipp32f);
    int dstStride0 = sizeof(Ipp32f)*3;  /* formally must be initialized */

    int length = 3;
    int count  = 1;

    IppStatus status = ippmCopy_va_32f_PS((const Ipp32f**)ppSrc,
        srcRoiShift, srcStride0, pDst, dstStride0,
        dstStride2, length, count );

    printf_va_Ipp32f("Destination vector:", pDst, 3, 1, status);
    return status;
}
```

The program above produces the following output:

```
Destination vector:
0.000000  6.000000  7.000000
```

The following example demonstrates how to use the function `ippmCopy_ma_32f_LS`. For more information, see also examples in the Getting Started chapter.

Example 3-2 ippmCopy_ma_32f_LS

```
IppStatus copy_ma_32f_LS(void) {
    /* Source data: 4 matrices with width=3 and height=2 */
    Ipp32f a[2*3] = { 10, 11, 12, 13, 14, 15 };
    Ipp32f b[2*3] = { 20, 21, 22, 23, 24, 25 };
    Ipp32f c[2*3] = { 30, 31, 32, 33, 34, 35 };
    Ipp32f d[2*3] = { 40, 41, 42, 43, 44, 45 };
    /*
```

Example 3-2 ippmCopy_ma_32f_LS (continued)

```
    // Layout description for 4 source matrices:
    */Ipp32f* ppSrc[4] = { a, b, c, d }; /* matrix pointers array */
    int srcRoiShift = 0;
    int srcStride2 = sizeof(Ipp32f);
    int srcStride1 = sizeof(Ipp32f)*3;

    /*
    // Standard description for 4 destination matrices
    */
    Ipp32f pDst[4*2*3];
    int dstStride2 = sizeof(Ipp32f);
    int dstStride1 = sizeof(Ipp32f)*3;
    int dstStride0 = sizeof(Ipp32f)*3*2;


    int width  = 3;
    int height = 2;
    int count  = 4;

    IppStatus status = ippmCopy_ma_32f_LS((const Ipp32f**)ppSrc,
        srcRoiShift, srcStride1, srcStride2, pDst, dstStride0,
        dstStride1, dstStride2, width, height, count );

    printf_va_Ipp32f("4 destination matrices:", pDst, 2*3, 4, status);
    return status;
}
```

The program above produces the following output:

```
4 destination matrices:
10.000000  11.000000  12.000000  13.000000  14.000000  15.000000
20.000000  21.000000  22.000000  23.000000  24.000000  25.000000
30.000000  31.000000  32.000000  33.000000  34.000000  35.000000
40.000000  41.000000  42.000000  43.000000  44.000000  45.000000
```

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the input size parameter is equal to 0.

ippStsStrideMatrixErr	Returns an error when the stride value is not positive or not divisible by size of data type.
ippStsRoiShiftMatrixErr	Returns an error when the roiShift value is negative or not divisible by size of data type.
ippStsCountMatrixErr	Returns an error when the count value is less or equal to zero.

Extract

Performs ROI extraction.

Syntax

Case 1: Vector operation

```
IppStatus ippmExtract_v_32f(const Ipp32f* pSrc, int srcStride2, Ipp32f* pDst,
    int len);
IppStatus ippmExtract_v_64f(const Ipp64f* pSrc, int srcStride2, Ipp64f* pDst,
    int len);
IppStatus ippmExtract_v_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    Ipp32f* pDst, int len);
IppStatus ippmExtract_v_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    Ipp64f* pDst, int len);
```

Case 2: Vector array operation

```
IppStatus, ippmExtract_va_32f(const Ipp32f* pSrc, int srcStride0,
    int srcStride2, Ipp32f* pDst, int len, int count);
IppStatus ippmExtract_va_64f(const Ipp64f* pSrc, int srcStride0,
    int srcStride2, Ipp64f* pDst, int len, int count);
IppStatus ippmExtract_va_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp32f* pDst, int len, int count);
IppStatus ippmExtract_va_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp64f* pDst, int len, int count);
IppStatus ippmExtract_va_32f_L(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride2, Ipp32f* pDst, int len, int count);
IppStatus ippmExtract_va_64f_L(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride2, Ipp64f* pDst, int len, int count);
```

Case 3: Matrix operation

```
IppStatus ippmExtract_m_32f(const Ipp32f* pSrc, int srcStride1, int srcStride2,
   Ipp32f* pDst, int width, int height);
IppStatus ippmExtract_m_64f(const Ipp64f* pSrc, int srcStride1, int srcStride2,
   Ipp64f* pDst, int width, int height);
IppStatus ippmExtract_m_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
   Ipp32f* pDst, int width, int height);
IppStatus ippmExtract_m_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
   Ipp64f* pDst, int width, int height);
```

Case 4: Transposed matrix operation

```
IppStatus ippmExtract_t_32f(const Ipp32f* pSrc, int srcStride1,
   int srcStride2, Ipp32f* pDst, int width, int height);
IppStatus ippmExtract_t_64f(const Ipp64f* pSrc, int srcStride1,
   int srcStride2, Ipp64f* pDst, int width, int height);
IppStatus ippmExtract_t_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
   Ipp32f* pDst, int width, int height);
IppStatus ippmExtract_t_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
   Ipp64f* pDst, int width, int height);
```

Case 5: Matrix array operation

```
IppStatus ippmExtract_ma_32f(const Ipp32f* pSrc, int srcStride0,
   int srcStride1, int srcStride2, Ipp32f* pDst, int width, int height,
   int count);
IppStatus ippmExtract_ma_64f(const Ipp64f* pSrc, int srcStride0,
   int srcStride1, int srcStride2, Ipp64f* pDst, int width, int height,
   int count);
IppStatus ippmExtract_ma_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
   int srcStride0, Ipp32f* pDst, int width, int height, int count);
IppStatus ippmExtract_ma_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
   int srcStride0, Ipp64f* pDst, int width, int height, int count);
IppStatus ippmExtract_ma_32f_L(const Ipp32f** ppSrc, int srcRoiShift,
   int srcStride1, int srcStride2, Ipp32f* pDst, int width, int height,
   int count);
IppStatus ippmExtract_ma_64f_L(const Ipp64f** ppSrc, int srcRoiShift,
   int srcStride1, int srcStride2, Ipp64f* pDst, int width, int height,
   int count);
```

Case 6: Transposed matrix array operation

```
IppStatus ippmExtract_ta_32f(const Ipp32f* pSrc, int srcStride0,
    int srcStride1, int srcStride2, Ipp32f* pDst, int width, int height,
    int count);
IppStatus ippmExtract_ta_64f(const Ipp64f* pSrc, int srcStride0,
    int srcStride1, int srcStride2, Ipp64f* pDst, int width, int height,
    int count);
IppStatus ippmExtract_ta_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp32f* pDst, int width, int height, int count);
IppStatus ippmExtract_ta_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp64f* pDst, int width, int height, int count);
IppStatus ippmExtract_ta_32f_L(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, Ipp32f* pDst, int width, int height,
    int count);
IppStatus ippmExtract_ta_64f_L(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, Ipp64f* pDst, int width, int height,
    int count);
```

Parameters

pSrc, ppSrc	Pointer to the source object or array of objects.
srcStride0	Stride between the objects in the source array.
srcStride1	Stride between the rows in the source matrix(ces).
srcStride2	Stride between the elements in the source object.
srcRoiShift	ROI shift in the source object.
pDst	Pointer to the specified destination object or array of objects.
len	Vector length.
width	Matrix width.
height	Matrix height.
count	Number of objects in the array.

Description

The function `ippmExtract` is declared in the `ippm.h` header file. The function extracts ROI from an object of any type to another object with specific properties.

When the operation is performed on vectors, the destination object is a dense vector or dense vector array.

When the operation is performed on matrices, the destination object is a dense matrix or a dense matrix array. The matrices involved in the operation must have the number of columns equal to *width* and the number of rows equal to *height*.

Note that if the operation is performed on a transposed matrix or an array of transposed matrices, the source matrices must have the number of columns equal to *height* and the number of rows equal to *width*.

Return Values

ippStsOk	Returns no error.
ippStsNullPtrErr	Returns an error when at least one input pointer is NULL.
ippStsSizeErr	Returns an error when the input size parameter is equal to 0.
ippStsStrideMatrixErr	Returns an error when the stride value is not positive or not divisible by size of data type.
ippStsRoiShiftMatrixErr	Returns an error when the roiShift value is negative or not divisible by size of data type.
ippStsCountMatrixErr	Returns an error when the count value is less or equal to zero.

LoadIdentity

Initializes identity matrix.

Syntax

Case 1: Matrix array operation

```
IppStatus ippmLoadIdentity_ma_32f(const Ipp32f* pDst, int dstStride0,
    int dstStride1, int dstStride2, int width, int height, int count);
IppStatus ippmLoadIdentity_ma_64f(const Ipp64f* pDst, int dstStride0,
    int dstStride1, int dstStride2, int width, int height, int count);
IppStatus ippmLoadIdentity_ma_32f_P(const Ipp32f** ppDst, int dstRoiShift,
    int dstStride2, int width, int height, int count);
IppStatus ippmLoadIdentity_ma_64f_P(const Ipp64f** ppDst, int dstRoiShift,
    int dstStride2, int width, int height, int count);
```

```
IppStatus ippmLoadIdentity_ma_32f_L(Ipp32f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int width, int height, int count);
IppStatus ippmLoadIdentity_ma_64f_L(Ipp64f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int width, int height, int count);
```

Parameters

pDst, ppDst	Pointer to the destination matrix or array of matrices.
dstStride0	Stride between the objects in the destination array.
srcStride1	Stride between the rows in the source matrix(ces).
dstStride2	Stride between the rows in the destination matrix(ces).
dstRoiShift	ROI shift in the destination matrix.
width	Matrix width.
height	Matrix height.
count	Number of objects in the array.

Description

The function `ippmLoadIdentity` is declared in the `ippm.h` header file. The function loads an identity matrix and stores the result in the destination object.

The destination matrix has the number of columns equal to *width* and the number of rows equal to *height*.

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the input size parameter is equal to 0.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not divisible by size of data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the roiShift value is negative or not divisible by size of data type.
`ippStsCountMatrixErr`	Returns an error when the count value is less or equal to zero.

Vector Algebra Functions 4

This chapter describes Intel® Integrated Performance Primitives (Intel® IPP) functions for small matrices that perform vector algebra operations.

Table 4-1 Vector Algebra functions

Function Base Name	Operation
Saxpy	Performs the “saxpy” operation on vectors.
Add	Adds constant to vector or vector to another vector.
Sub	Subtracts constant from vector, vector from constant, or vector from another vector.
Mul	Multiplies vector by constant.
CrossProduct	Computes cross product of two 3D vectors.
DotProduct	Computes dot product of two vectors.
L2Norm	Computes vector’s L2 norm.
LComb	Composes linear combination of two vectors.

Saxpy

Performs the “saxpy” operation on vectors.

Syntax

Case 1: Vector - vector operation

```
IppStatus ippmSaxpy_vv_32f(const Ipp32f* pSrc1, int src1Stride2, Ipp32f scale,
    const Ipp32f* pSrc2, int src2Stride2, Ipp32f* pDst, int dstStride2,
    int len);
```

```
IppStatus ippmSaxpy_vv_64f(const Ipp64f* pSrc1, int src1Stride2, Ipp64f scale,
    const Ipp64f* pSrc2, int src2Stride2, Ipp64f* pDst, int dstStride2,
    int len);

IppStatus ippmSaxpy_vv_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    Ipp32f scale, const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst,
    int dstRoiShift, int len);

IppStatus ippmSaxpy_vv_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    Ipp64f scale, const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst,
    int dstRoiShift, int len);
```

Case 2: Vector - vector array operation

```
IppStatus ippmSaxpy_vva_32f(const Ipp32f* pSrc1, int src1Stride2, Ipp32f scale,
    const Ipp32f* pSrc2, int src2Stride0, int src2Stride2, Ipp32f* pDst,
    int dstStride0, int dstStride2, int len, int count);

IppStatus ippmSaxpy_vva_64f(const Ipp64f* pSrc1, int src1Stride2, Ipp64f scale,
    const Ipp64f* pSrc2, int src2Stride0, int src2Stride2, Ipp64f* pDst,
    int dstStride0, int dstStride2, int len, int count);

IppStatus ippmSaxpy_vva_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    Ipp32f scale, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp32f** ppDst, int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmSaxpy_vva_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    Ipp64f scale, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp64f** ppDst, int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmSaxpy_vva_32f_L(const Ipp32f* pSrc1, int src1Stride2,
    Ipp32f scale, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2,
    Ipp32f** ppDst, int dstRoiShift, int dstStride2, int len, int count);

IppStatus ippmSaxpy_vva_64f_L(const Ipp64f* pSrc1, int src1Stride2,
    Ipp64f scale, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2,
    Ipp64f** ppDst, int dstRoiShift, int dstStride2, int len, int count);
```

Case 3: Vector array - vector operation

```
IppStatus ippmSaxpy_vav_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride2, Ipp32f scale, const Ipp32f* pSrc2, int src2Stride2,
    Ipp32f* pDst, int dstStride0, int dstStride2, int len, int count);

IppStatus ippmSaxpy_vav_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride2, Ipp64f scale, const Ipp64f* pSrc2, int src2Stride2,
    Ipp64f* pDst, int dstStride0, int dstStride2, int len, int count);

IppStatus ippmSaxpy_vav_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, Ipp32f scale, const Ipp32f** ppSrc2, int src2RoiShift,
    Ipp32f** ppDst, int dstRoiShift, int dstStride0, int len, int count);
```

```
IppStatus ippmSaxpy_vav_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, Ipp64f scale, const Ipp64f** ppSrc2, int src2RoiShift,
    Ipp64f** ppDst, int dstRoiShift, int dstStride0, int len, int count);
IppStatus ippmSaxpy_vav_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride2, Ipp32f scale, const Ipp32f* pSrc2, int src2Stride2,
    Ipp32f** ppDst, int dstRoiShift, int dstStride2, int len, int count);
IppStatus ippmSaxpy_vav_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride2, Ipp64f scale, const Ipp64f* pSrc2, int src2Stride2,
    Ipp64f** ppDst, int dstRoiShift, int dstStride2, int len, int count);
```

Case 4: Vector array - vector array operation

```
IppStatus ippmSaxpy_vava_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride2, Ipp32f scale, const Ipp32f* pSrc2, int src2Stride0,
    int src2Stride2, Ipp32f* pDst, int dstStride0, int dstStride2, int len,
    int count);
IppStatus ippmSaxpy_vava_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride2, Ipp64f scale, const Ipp64f* pSrc2, int src2Stride0,
    int src2Stride2, Ipp64f* pDst, int dstStride0, int dstStride2, int len,
    int count);
IppStatus ippmSaxpy_vava_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, Ipp32f scale, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Stride0, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int len,
    int count);
IppStatus ippmSaxpy_vava_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, Ipp64f scale, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Stride0, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int len,
    int count);
IppStatus ippmSaxpy_vava_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride2, Ipp32f scale, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride2, int len,
    int count);
IppStatus ippmSaxpy_vava_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride2, Ipp64f scale, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride2, int len,
    int count);
```

Parameters

pSrc1, ppSrc1	Pointer to the first source vector or array.
src1Stride0	Stride between the vectors in the first source vector array.

src1Stride2	Stride between the elements in the first source vector(s).
src1RoiShift	ROI shift in the first source vector(s).
pSrc2, ppSrc2	Pointer to the second source vector or vector array.
src2Stride0	Stride between the vectors in the second source vector array.
src2Stride2	Stride between the elements in the second source vector(s).
src2RoiShift	ROI shift in the second source vector(s).
pDst, ppDst	Pointer to the destination vector or vector array.
dstStride0	Stride between the vectors in the destination array.
dstStride2	Stride between the elements in the destination vector(s).
dstRoiShift	ROI shift in the destination vector(s).
scale	Multiplier.
len	Vector length.
count	Number of vectors in the array.

Description

The function `ippmSaxpy` is declared in the `ippm.h` header file. The function composes linear combination of two vectors by multiplying the first source vector by a constant, adding it to the second vector, and storing the result in the destination vector:

$dst[i] = scale \times src1[i] + src2[i],$

$0 \le i < len.$

The following example demonstrates how to use the function `ippmSaxpy_vav_32f`. For more information, see also examples in the Getting Started chapter.

Example 4-1 ippmSaxpy_vav_32f

```
IppStatus saxpy_vav_32f(void) {
    /* Src1 is 2 vectors with length=4 */
    Ipp32f pSrc1[2*4] = { 1, 2, 4, 8,
                          3, 5, 7, 9};
    /* Src2 is vector with length=4 */
    Ipp32f pSrc2[4] = { -1, -5, -2 , -3 };

    Ipp32f scale = 2.0;
    /* Standard description for source vectors */
    int src1Stride2 = sizeof(Ipp32f);
    int src1Stride0 = 4*sizeof(Ipp32f);
    int src2Stride2 = sizeof(Ipp32f);

    /* Standard description for destination vectors */
    Ipp32f pDst[2*4];
    int dstStride2 = sizeof(Ipp32f);
    int dstStride0 = 4*sizeof(Ipp32f);

    int length = 4;
    int count  = 2;

    IppStatus status = ippmSaxpy_vav_32f((const Ipp32f*)pSrc1,
        src1Stride0, src1Stride2, scale,(const Ipp32f*)pSrc2,
         src2Stride2, pDst, dstStride0, dstStride2, length, count);

    printf_va_Ipp32f("Dst is 2 vectors:", pDst, 4, 2, status);

    return status;
}
```

The program above produces the following output:

```
Dst is 2 vectors:
1.000000  -1.000000  6.000000  13.000000
5.000000  5.000000  12.000000  15.000000
```

Return Values

`ippStsOk`	Returns no error.

ippStsNullPtrErr	Returns an error when at least one input pointer is NULL.
ippStsSizeErr	Returns an error when the input size parameter is equal to 0.
ippStsStrideMatrixErr	Returns an error when the stride value is not positive or not divisible by size of data type.
ippStsRoiShiftMatrixErr	Returns an error when the roiShift value is negative or not divisible by size of data type.
ippStsCountMatrixErr	Returns an error when the count value is less or equal to zero.

Add

Adds constant to vector or vector to another vector.

Syntax

Case 1: Vector - constant operation

```
IppStatus ippmAdd_vc_32f(const Ipp32f* pSrc, int srcStride2, Ipp32f val,
    Ipp32f* pDst, int dstStride2, int len);
IppStatus ippmAdd_vc_64f(const Ipp64f* pSrc, int srcStride2, Ipp64f val,
    Ipp64f* pDst, int dstStride2, int len);
IppStatus ippmAdd_vc_32f_P(const Ipp32f** ppSrc, int srcRoiShift, Ipp32f val,
    Ipp32f** ppDst, int dstRoiShift, int len);
IppStatus ippmAdd_vc_64f_P(const Ipp64f** ppSrc, int srcRoiShift, Ipp64f val,
    Ipp64f** ppDst, int dstRoiShift, int len);
```

Case 2: Vector array - constant operation

```
IppStatus ippmAdd_vac_32f(const Ipp32f* pSrc, int srcStride0, int srcStride2,
    Ipp32f val, Ipp32f* pDst, int dstStride0, int dstStride2, int len,
    int count);
IppStatus ippmAdd_vac_64f(const Ipp64f* pSrc, int srcStride0, int srcStride2,
    Ipp64f val, Ipp64f* pDst, int dstStride0, int dstStride2, int len,
    int count);
IppStatus ippmAdd_vac_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp32f val, Ipp32f** ppDst, int dstRoiShift, int dstStride0,
    int len, int count);
```

```
IppStatus ippmAdd_vac_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp64f val, Ipp64f** ppDst, int dstRoiShift, int dstStride0,
    int len, int count);

IppStatus ippmAdd_vac_32f_L(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride2, Ipp32f val, Ipp32f** ppDst, int dstRoiShift, int dstStride2,
    int len, int count);

IppStatus ippmAdd_vac_64f_L(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride2, Ipp64f val, Ipp64f** ppDst, int dstRoiShift, int dstStride2,
    int len, int count);
```

Case 3: Vector array - vector operation

```
IppStatus ippmAdd_vv_32f(const Ipp32f* pSrc1, int src1Stride2,
    const Ipp32f* pSrc2, int src2Stride2, Ipp32f* pDst, int dstStride2,
    int len);

IppStatus ippmAdd_vv_64f(const Ipp64f* pSrc1, int src1Stride2,
    const Ipp64f* pSrc2, int src2Stride2, Ipp64f* pDst, int dstStride2,
    int len);

IppStatus ippmAdd_vv_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst, int dstRoiShift,
    int len);

IppStatus ippmAdd_vv_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst, int dstRoiShift,
    int len);
```

Case 3: Vector array - vector operation

```
IppStatus ippmAdd_vav_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride2, const Ipp32f* pSrc2, int src2Stride2, Ipp32f* pDst,
    int dstStride0, int dstStride2, int len, int count);

IppStatus ippmAdd_vav_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride2, const Ipp64f* pSrc2, int src2Stride2, Ipp64f* pDst,
    int dstStride0, int dstStride2, int len, int count);

IppStatus ippmAdd_vav_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst,
    int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmAdd_vav_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst,
    int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmAdd_vav_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride2, const Ipp32f* pSrc2, int src2Stride2, Ipp32f** ppDst,
    int dstRoiShift, int dstStride2, int len, int count);
```

```
IppStatus ippmAdd_vav_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride2, const Ipp64f* pSrc2, int src2Stride2, Ipp64f** ppDst,
    int dstRoiShift, int dstStride2, int len, int count);
```

Case 4: Vector array - vector array operation

```
IppStatus ippmAdd_vava_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride2, const Ipp32f* pSrc2, int src2Stride0, int src2Stride2,
    Ipp32f* pDst, int dstStride0, int dstStride2, int len, int count);

IppStatus ippmAdd_vava_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride2, const Ipp64f* pSrc2, int src2Stride0, int src2Stride2,
    Ipp64f* pDst, int dstStride0, int dstStride2, int len, int count);

IppStatus ippmAdd_vava_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp32f** ppDst, int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmAdd_vava_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp64f** ppDst, int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmAdd_vava_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2,
    Ipp32f** ppDst, int dstRoiShift, int dstStride2, int len, int count);

IppStatus ippmAdd_vava_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2,
    Ipp64f** ppDst, int dstRoiShift, int dstStride2, int len, int count);
```

Parameters

pSrc, ppSrc	Pointer to the source vector or vector array.
srcStride0	Stride between the vectors in the source array.
srcStride2	Stride between the elements in the source vector(s).
srcRoiShift	ROI shift in the first source vector.
pSrc1, ppSrc1	Pointer to the first source vector or vector array.
src1Stride0	Stride between the vectors in the first source vector array.
src1Stride2	Stride between the elements in the first source vector(s).
src1RoiShift	ROI shift in the first source vector.
pSrc2, ppSrc2	Pointer to the second source vector or vector array.
src2Stride0	Stride between the vectors in the second source vector array.

src2Stride2	Stride between the elements in the second source vector(s).
src2RoiShift	ROI shift in the second source vector(s).
pDst, ppDst	Pointer to the destination vector or vector array.
dstStride0	Stride between the vectors in the destination array.
dstStride2	Stride between the elements in the destination vector.
dstRoiShift	ROI shift in the destination vector.
val	Added value.
len	Vector length.
count	Number of vectors in the array.

Description

The function `ippmAdd` is declared in the `ippm.h` header file. Like all other Intel IPP matrix operating functions, this function is parameter sensitive. All input parameters that follow the function name immediately after the underscore determine the way in which the function performs and the arguments it takes, whether it is a constant or another vector. This implies that with every complete function name only some of the listed arguments appear in the input list, while others are omitted.

When performed on a constant together with a vector, the function adds *val* to each element of the source vector and stores the result into destination vector:

$dst[i] = val + src[i],$

$0 \le i < len.$

The following example demonstrates how to use the function `ippmAdd_vc_32f_P`. For more information, see also examples in the Getting Started chapter.

Example 4-2 ippmAdd_vc_32f_P

```
IppStatus add_vc_32f_P(void) {
    /* Source data:  */
    Ipp32f a[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    Ipp32f val = 10.0;
    /*
    // Pointer description for interested source data a[0], a[6], a[7]:
    */
    Ipp32f* ppSrc[3] = { a, a+6, a+7 }; /* pointers array */
    int srcRoiShift = 0;

    /*
    // Pointer description for interested destination data a[0], a[6], a[7]:
    */
    Ipp32f* ppDst[3] = { a, a+6, a+7 }; /* pointers array */
    int dstRoiShift  = 0;

    int length = 3;

    IppStatus status = ippmAdd_vc_32f_P((const Ipp32f**)ppSrc,
        srcRoiShift, val, ppDst, dstRoiShift, length);

    printf_va_Ipp32f("Destination vector:", a, 10, 1, status);
    return status;
}
```

The program above produces the following output:

```
Destination vector:

10.000000  1.000000  2.000000  3.000000  4.000000  5.000000  16.000000
17.000000  8.000000  9.000000
```

When performed on two vectors, the function adds together the respective elements of the first and the second source vectors and stores the result in the destination vector:

$dst[i] = src1[i] + src2[i]$,

$0 \le i < len$.

The following example demonstrates how to use the function `ippmAdd_vava_32f_L`. For more information, see also examples in the Getting Started chapter.

Example 4-3 ippmAdd_vava_32f_L

```
IppStatus add_vava_32f_L(void) {
    /* Src1 data: 4 vectors with length=3, Stride2=2*sizeof(Ipp32f) */
    Ipp32f src1_a[2*3] = { 1, 0, 2, 0, 3, 0 };
    Ipp32f src1_b[2*3] = { 4, 0, 5, 0, 6, 0 };
    Ipp32f src1_c[2*3] = { 7, 0, 8, 0, 9, 0 };
    Ipp32f src1_d[2*3] = { 10, 0, 11, 0, 12, 0 };
  /* Src2 data: 4 vectors with length=3, Stride2=sizeof(Ipp32f) */
    Ipp32f src2_a[3] = { 10, 11, 12 };
    Ipp32f src2_b[3] = { 7, 8, 9 };
    Ipp32f src2_c[3] = { 4, 5, 6 };
    Ipp32f src2_d[3] = { 1, 2, 3 };

    /* Layout description for Src1, Src2 and Dst */
    Ipp32f* ppSrc1[4] = { src1_a, src1_b, src1_c, src1_d };
    Ipp32f* ppSrc2[4] = { src2_d, src2_c, src2_b, src2_a };
    int src1RoiShift = 0;
    int src2RoiShift = 0;

    Ipp32f dst[4*3]; /* destination memory location */
    Ipp32f* ppDst[4] = { dst, dst+3, dst+6, dst+9 };
    int dstRoiShift  = 0;

    int src1Stride2 = 2*sizeof(Ipp32f);
    int src2Stride2 = sizeof(Ipp32f);
    int dstStride2  = sizeof(Ipp32f);

    int length = 3;
    int count  = 4;

    IppStatus status = ippmAdd_vava_32f_L((const Ipp32f**)ppSrc1,
        src1RoiShift, src1Stride2, (const Ipp32f**)ppSrc2, src2RoiShift,
        src2Stride2, ppDst, dstRoiShift, dstStride2, length, count);

    printf_va_Ipp32f("4 destination vectors:", dst, 3, 4, status);
    return status;
}
```

The program above produces the following output:

```
4 destination vectors:
2.000000  4.000000  6.000000
8.000000  10.000000  12.000000
14.000000  16.000000  18.000000
20.000000  22.000000  24.000000
```

Return Values

ippStsOk	Returns no error.
ippStsNullPtrErr	Returns an error when at least one input pointer is NULL.
ippStsSizeErr	Returns an error when the input size parameter is equal to 0.
ippStsStrideMatrixErr	Returns an error when the stride value is not positive or not divisible by size of data type.
ippStsRoiShiftMatrixErr	Returns an error when the roiShift value is negative or not divisible by size of data type.
ippStsCountMatrixErr	Returns an error when the count value is less or equal to zero.

Sub

Subtracts constant from vector, vector from constant, or vector from another vector.

Syntax

Case 1: Vector - constant operation

```
IppStatus ippmSub_vc_32f(const Ipp32f* pSrc, int srcStride2, Ipp32f val,
    Ipp32f* pDst, int dstStride2, int len);
IppStatus ippmSub_vc_64f(const Ipp64f* pSrc, int srcStride2, Ipp64f val,
    Ipp64f* pDst, int dstStride2, int len);
IppStatus ippmSub_vc_32f_P(const Ipp32f** ppSrc, int srcRoiShift, Ipp32f val,
    Ipp32f** ppDst, int dstRoiShift, int len);
IppStatus ippmSub_vc_64f_P(const Ipp64f** ppSrc, int srcRoiShift, Ipp64f val,
    Ipp64f** ppDst, int dstRoiShift, int len);
```

Case 2: Vector array - constant operation

```
IppStatus ippmSub_vac_32f(const Ipp32f* pSrc, int srcStride0, int srcStride2,
    Ipp32f val, Ipp32f* pDst, int dstStride0, int dstStride2, int len,
    int count);
IppStatus ippmSub_vac_64f(const Ipp64f* pSrc, int srcStride0, int srcStride2,
    Ipp64f val, Ipp64f* pDst, int dstStride0, int dstStride2, int len,
    int count);
IppStatus ippmSub_vac_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp32f val, Ipp32f** ppDst, int dstRoiShift, int dstStride0,
    int len, int count);
IppStatus ippmSub_vac_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp64f val, Ipp64f** ppDst, int dstRoiShift, int dstStride0,
    int len, int count);
IppStatus ippmSub_vac_32f_L(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride2, Ipp32f val, Ipp32f** ppDst, int dstRoiShift, int dstStride2,
    int len, int count);
IppStatus ippmSub_vac_64f_L(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride2, Ipp64f val, Ipp64f** ppDst, int dstRoiShift, int dstStride2,
    int len, int count);
```

Case 3: Constant - vector operation

```
IppStatus ippmSub_cv_32f(const Ipp32f* pSrc, int srcStride2, Ipp32f val,
    Ipp32f* pDst, int dstStride2, int len);
IppStatus ippmSub_cv_64f(const Ipp64f* pSrc, int srcStride2, Ipp64f val,
    Ipp64f* pDst, int dstStride2, int len);
IppStatus ippmSub_cv_32f_P(const Ipp32f** ppSrc, int srcRoiShift, Ipp32f val,
    Ipp32f** ppDst, int dstRoiShift, int len);
IppStatus ippmSub_cv_64f_P(const Ipp64f** ppSrc, int srcRoiShift, Ipp64f val,
    Ipp64f** ppDst, int dstRoiShift, int len);
```

Case 4: Constant - vector array operation

```
IppStatus ippmSub_cva_32f(const Ipp32f* pSrc, int srcStride0, int srcStride2,
    Ipp32f val, Ipp32f* pDst, int dstStride0, int dstStride2, int len,
    int count);
IppStatus ippmSub_cva_64f(const Ipp64f* pSrc, int srcStride0, int srcStride2,
    Ipp64f val, Ipp64f* pDst, int dstStride0, int dstStride2, int len,
    int count);
IppStatus ippmSub_cva_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp32f val, Ipp32f** ppDst, int dstRoiShift, int dstStride0,
    int len, int count);
```

```
IppStatus ippmSub_cva_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp64f val, Ipp64f** ppDst, int dstRoiShift, int dstStride0,
    int len, int count);

IppStatus ippmSub_cva_32f_L(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride2, Ipp32f val, Ipp32f** ppDst, int dstRoiShift, int dstStride2,
    int len, int count);

IppStatus ippmSub_cva_64f_L(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride2, Ipp64f val, Ipp64f** ppDst, int dstRoiShift, int dstStride2,
    int len, int count);
```

Case 5: Vector - vector operation

```
IppStatus ippmSub_vv_32f(const Ipp32f* pSrc1, int src1Stride2,
    const Ipp32f* pSrc2, int src2Stride2, Ipp32f* pDst, int dstStride2,
    int len);

IppStatus ippmSub_vv_64f(const Ipp64f* pSrc1, int src1Stride2,
    const Ipp64f* pSrc2, int src2Stride2, Ipp64f* pDst, int dstStride2,
    int len);

IppStatus ippmSub_vv_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst, int dstRoiShift,
    int len);

IppStatus ippmSub_vv_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst, int dstRoiShift,
    int len);
```

Case 6: Vector - vector array operation

```
IppStatus ippmSub_vva_32f(const Ipp32f* pSrc1, int src1Stride2,
    const Ipp32f* pSrc2, int src2Stride0, int src2Stride2, Ipp32f* pDst,
    int dstStride0, int dstStride2, int len, int count);

IppStatus ippmSub_vva_64f(const Ipp64f* pSrc1, int src1Stride2,
    const Ipp64f* pSrc2, int src2Stride0, int src2Stride2, Ipp64f* pDst,
    int dstStride0, int dstStride2, int len, int count);

IppStatus ippmSub_vva_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp32f** ppDst,
    int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmSub_vva_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp64f** ppDst,
    int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmSub_vva_32f_L(const Ipp32f* pSrc1, int src1Stride2,
    const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2, Ipp32f** ppDst,
    int dstRoiShift, int dstStride2, int len, int count);
```

```
IppStatus ippmSub_vva_64f_L(const Ipp64f* pSrc1, int src1Stride2,
   const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2, Ipp64f** ppDst,
   int dstRoiShift, int dstStride2, int len, int count);
```

Case 7: Vector array - vector operation

```
IppStatus ippmSub_vav_32f(const Ipp32f* pSrc1, int src1Stride0,
   int src1Stride2, const Ipp32f* pSrc2, int src2Stride2, Ipp32f* pDst,
   int dstStride0, int dstStride2, int len, int count);

IppStatus ippmSub_vav_64f(const Ipp64f* pSrc1, int src1Stride0,
   int src1Stride2, const Ipp64f* pSrc2, int src2Stride2, Ipp64f* pDst,
   int dstStride0, int dstStride2, int len, int count);

IppStatus ippmSub_vav_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
   int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst,
   int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmSub_vav_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
   int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst,
   int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmSub_vav_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
   int src1Stride2, const Ipp32f* pSrc2, int src2Stride2, Ipp32f** ppDst,
   int dstRoiShift, int dstStride2, int len, int count);

IppStatus ippmSub_vav_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
   int src1Stride2, const Ipp64f* pSrc2, int src2Stride2, Ipp64f** ppDst,
   int dstRoiShift, int dstStride2, int len, int count);
```

Case 8: Vector array - vector array operation

```
IppStatus ippmSub_vava_32f(const Ipp32f* pSrc1, int src1Stride0,
   int src1Stride2, const Ipp32f* pSrc2, int src2Stride0, int src2Stride2,
   Ipp32f* pDst, int dstStride0, int dstStride2, int len, int count);

IppStatus ippmSub_vava_64f(const Ipp64f* pSrc1, int src1Stride0,
   int src1Stride2, const Ipp64f* pSrc2, int src2Stride0, int src2Stride2,
   Ipp64f* pDst, int dstStride0, int dstStride2, int len, int count);

IppStatus ippmSub_vava_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
   int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0,
   Ipp32f** ppDst, int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmSub_vava_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
   int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0,
   Ipp64f** ppDst, int dstRoiShift, int dstStride0, int len, int count);

IppStatus ippmSub_vava_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
   int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2,
   Ipp32f** ppDst, int dstRoiShift, int dstStride2, int len, int count);
```

```
IppStatus ippmSub_vava_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2,
    Ipp64f** ppDst, int dstRoiShift, int dstStride2, int len, int count);
```

Parameters

pSrc, ppSrc	Pointer to the source vector or vector array.
srcStride0	Stride between the vectors in the source array.
srcStride2	Stride between the elements in the source vector(s).
srcRoiShift	ROI shift in the source vector.
pSrc1, ppSrc1	Pointer to the first source vector or vector array.
src1Stride0	Stride between the vectors in the first source vector array.
src1Stride2	Stride between the elements in the first source vector(s).
src1RoiShift	ROI shift in the first source vector.
pSrc2, ppSrc2	Pointer to the second source vector or vector array.
src2Stride0	Stride between the vectors in the second source vector array.
src2Stride2	Stride between the elements in the second source vector(s).
src2RoiShift	ROI shift in the second source vector(s).
pDst, ppDst	Pointer to the destination vector or vector array.
dstStride0	Stride between the vectors in the destination array.
dstStride2	Stride between the elements in the destination vector.
dstRoiShift	ROI shift in the destination vector.
val	Subtracted value.
len	Vector length.
count	Number of vectors in the array.

Description

The function `ippmSub` is declared in the `ippm.h` header file. Like all other Intel IPP matrix operating functions, this function is parameter sensitive. All input parameters that follow the function name immediately after the underscore determine the way in which the function performs

and the arguments it takes, whether it is a constant or another vector. This implies that with every complete function name only some of the listed arguments appear in the input list, while others are omitted.

When the function is performed on a vector and a constant, it subtracts *val* from each element of the source vector and stores the result in the destination vector:

$$dst[i] = src[i] - val,$$

$$0 \le i < len.$$

When the function is performed on a constant and a vector, it subtracts each element of the source vector from *val* and stores the result in the destination vector:

$$dst[i] = val - src[i],$$

$$0 \le i < len.$$

When the function is performed on two vectors, it subtracts the elements of the second source vector from the respective elements of the first source vector and stores the result in the destination vector:

$$dst[i] = src1[i] - src2[i],$$

$$0 \le i < len.$$

The following example demonstrates how to use the function `ippmSub_vav_32f_P`. For more information, see also examples in the Getting Started chapter.

Example 4-4 ippmSub_vav_32f_P

```
IppStatus sub_vav_32f_P(void) {
    /* Source data */
    Ipp32f src1[3*4] = { 11, 21, 31, 41,
                         12, 22, 32, 42,
                         13, 23, 33, 43 };

    Ipp32f src2[3] = { 1, 2, 3 };
    /*
    // The first operand is 4 vector-columns with length=3
    // Pointer description:
    // ppSrc1[0] pointer to the first element of the first column
    // ppSrc1[1] pointer to the second element of the first column
    // ppSrc1[2] pointer to the third element  of the first column
    */
```

Example 4-4 ippmSub_vav_32f_P (continued)

```
    Ipp32f* ppSrc1[3] = { src1, src1+4, src1+8 };
    int src1RoiShift = 0;
    int src1Stride0 = sizeof(Ipp32f); /* Stride between columns */

    /*
    // The second operand is vector with length=3
    // Pointer description:
    // ppSrc2[0] pointer to the first element
    // ppSrc2[1] pointer to the second element
    // ppSrc2[2] pointer to the third element
    */
    Ipp32f* ppSrc2[3] = { src2, src2+1, src2+2 };
    int src2RoiShift = 0;

    /*
    // Destination is 4 vectors with length=3
    // Pointer description for destination vectors
    */
    Ipp32f dst[12];
    int length = 3;
    int count  = 4;

    Ipp32f* ppDst[3] = { dst, dst+1, dst+2 };
    int dstRoiShift = 0;
    int dstStride0 = sizeof(Ipp32f)*3; /* Stride between vectors */

    IppStatus status = ippmSub_vav_32f_P((const Ipp32f**)ppSrc1,
        src1RoiShift, src1Stride0, (const Ipp32f**)ppSrc2, src2RoiShift,
        ppDst, dstRoiShift, dstStride0, length, count);
    printf_va_Ipp32f("Dst is 4 vectors:", dst, 3, 4, status);

    return status;
}
```

The program above produces the following output:

```
Dst is 4 vectors:
10.000000  10.000000  10.000000
20.000000  20.000000  20.000000
30.000000  30.000000  30.000000
40.000000  40.000000  40.000000
```

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the input size parameter is equal to 0.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not divisible by size of data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the roiShift value is negative or not divisible by size of data type.
`ippStsCountMatrixErr`	Returns an error when the count value is less or equal to zero.

Mul

Multiplies vector by constant.

Syntax

Case 1: Vector - constant operation

```
IppStatus ippmMul_vc_32f(const Ipp32f* pSrc, int srcStride2, Ipp32f val,
    Ipp32f* pDst, int dstStride2, int len);
IppStatus ippmMul_vc_64f(const Ipp64f* pSrc, int srcStride2, Ipp64f val,
    Ipp64f* pDst, int dstStride2, int len);
IppStatus ippmMul_vc_32f_P(const Ipp32f** ppSrc, int srcRoiShift, Ipp32f val,
    Ipp32f** ppDst, int dstRoiShift, int len);
IppStatus ippmMul_vc_64f_P(const Ipp64f** ppSrc, int srcRoiShift, Ipp64f val,
    Ipp64f** ppDst, int dstRoiShift, int len);
```

Case 2: Vector array - constant operation

```
IppStatus ippmMul_vac_32f(const Ipp32f* pSrc, int srcStride0, int srcStride2,
    Ipp32f val, Ipp32f* pDst, int dstStride0, int dstStride2, int len,
    int count);
IppStatus ippmMul_vac_64f(const Ipp64f* pSrc, int srcStride0, int srcStride2,
    Ipp64f val, Ipp64f* pDst, int dstStride0, int dstStride2, int len,
    int count);
```

```
IppStatus ippmMul_vac_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp32f val, Ipp32f** ppDst, int dstRoiShift, int dstStride0,
    int len, int count);
IppStatus ippmMul_vac_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp64f val, Ipp64f** ppDst, int dstRoiShift, int dstStride0,
    int len, int count);
IppStatus ippmMul_vac_32f_L(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride2, Ipp32f val, Ipp32f** ppDst, int dstRoiShift, int dstStride2,
    int len, int count);
IppStatus ippmMul_vac_64f_L(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride2, Ipp64f val, Ipp64f** ppDst, int dstRoiShift, int dstStride2,
    int len, int count);
```

Parameters

pSrc, ppSrc	Pointer to the source vector or vector array.
srcStride0	Stride between the vectors in the source array.
srcStride2	Stride between the elements in the source vector(s).
srcRoiShift	ROI shift in the source vector.
pDst, ppDst	Pointer to the destination vector or vector array.
dstStride0	Stride between the vectors in the destination array.
dstStride2	Stride between the elements in the destination vector.
dstRoiShift	ROI shift in the destination vector.
val	Multiplier.
len	Vector length.
count	Number of vectors in the array.

Description

The function `ippmMul` is declared in the `ippm.h` header file. The function multiplies the elements of the source vector by a constant and stores the result in the destination vector:

$dst[i] = val \times src[i]$,

$0 \le i < len$.

The following example demonstrates how to use the function `ippmMul_vac_32f`. For more information, see also examples in the Getting Started chapter.

Example 4-5 ippmMul_vac_32f

```
IppStatus mul_vac_32f(void){
    /* Source data */
    Ipp32f pSrc[2*6] = { 2, 1, 3, 1, 4, 1,
                         5, 1, 6, 1, 7, 1 };
    /*
    // Interested elements: 2 vectors with length=3
    */
    int srcStride2 = 2*sizeof(Ipp32f);
    int srcStride0 = 6*sizeof(Ipp32f);

    Ipp32f pDst[6*2] = {0};
    int dstStride2 = 2*sizeof(Ipp32f);
    int dstStride0 = 6*sizeof(Ipp32f);

    Ipp32f val=2.0;
    int length = 3;
    int count  = 2;

    IppStatus status = ippmMul_vac_32f((const Ipp32f*) pSrc, srcStride0,
        srcStride2, val, pDst, dstStride0, dstStride2, length, count);

    printf_va_Ipp32f("2 source vectors:", pSrc, 6, 2, status);
    printf_va_Ipp32f("2 destination vectors:", pDst, 6, 2, status);

    return status;
}
```

The program above produces the following output:

```
2 source vectors:
2.000000  1.000000  3.000000  1.000000  4.000000  1.000000
5.000000  1.000000  6.000000  1.000000  7.000000  1.000000

2 destination vectors:
4.000000  0.000000  6.000000  0.000000  8.000000  0.000000
10.000000  0.000000  12.000000  0.000000  14.000000  0.000000
```

Return Values

ippStsOk	Returns no error.
ippStsNullPtrErr	Returns an error when at least one input pointer is NULL.
ippStsSizeErr	Returns an error when the input size parameter is equal to 0.
ippStsStrideMatrixErr	Returns an error when the stride value is not positive or not divisible by size of data type.
ippStsRoiShiftMatrixErr	Returns an error when the roiShift value is negative or not divisible by size of data type.
ippStsCountMatrixErr	Returns an error when the count value is less or equal to zero.

CrossProduct

Computes cross product of two 3D vectors.

Syntax

Case 1: Vector - vector operation

```
IppStatus ippmCrossProduct_vv_32f(const Ipp32f* pSrc1, int src1Stride2,
    const Ipp32f* pSrc2, int src2Stride2, Ipp32f* pDst, int dstStride2);
IppStatus ippmCrossProduct_vv_64f(const Ipp64f* pSrc1, int src1Stride2,
    const Ipp64f* pSrc2, int src2Stride2, Ipp64f* pDst, int dstStride2);
IppStatus ippmCrossProduct_vv_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst, int dstRoiShift);
IppStatus ippmCrossProduct_vv_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst, int dstRoiShift);
```

Case 2: Vector - vector array operation

```
IppStatus ippmCrossProduct_vva_32f(const Ipp32f* pSrc1, int src1Stride2,
    const Ipp32f* pSrc2, int src2Stride0, int src2Stride2, Ipp32f* pDst,
    int dstStride0, int dstStride2, int count);
IppStatus ippmCrossProduct_vva_64f(const Ipp64f* pSrc1, int src1Stride2,
    const Ipp64f* pSrc2, int src2Stride0, int src2Stride2, Ipp64f* pDst,
    int dstStride0, int dstStride2, int count);
```

```
IppStatus ippmCrossProduct_vva_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
   const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp32f** ppDst,
   int dstRoiShift, int dstStride0, int count);

IppStatus ippmCrossProduct_vva_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
   const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp64f** ppDst,
   int dstRoiShift, int dstStride0, int count);

IppStatus ippmCrossProduct_vva_32f_L(const Ipp32f* pSrc1, int src1Stride2,
   const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2, Ipp32f** ppDst,
   int dstRoiShift, int dstStride2, int count);

IppStatus ippmCrossProduct_vva_64f_L(const Ipp64f* pSrc1, int src1Stride2,
   const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2, Ipp64f** ppDst,
   int dstRoiShift, int dstStride2, int count);
```

Case 3: Vector array - vector operation

```
IppStatus ippmCrossProduct_vav_32f(const Ipp32f* pSrc1, int src1Stride0,
   int src1Stride2, const Ipp32f* pSrc2, int src2Stride2, Ipp32f* pDst,
   int dstStride0, int dstStride2, int count);

IppStatus ippmCrossProduct_vav_64f(const Ipp64f* pSrc1, int src1Stride0,
   int src1Stride2, const Ipp64f* pSrc2, int src2Stride2, Ipp64f* pDst,
   int dstStride0, int dstStride2, int count);

IppStatus ippmCrossProduct_vav_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
   int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst,
   int dstRoiShift, int dstStride0, int count);

IppStatus ippmCrossProduct_vav_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
   int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst,
   int dstRoiShift, int dstStride0, int count);

IppStatus ippmCrossProduct_vav_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
   int src1Stride2, const Ipp32f* pSrc2, int src2Stride2, Ipp32f** ppDst,
   int dstRoiShift, int dstStride2, int count);

IppStatus ippmCrossProduct_vav_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
   int src1Stride2, const Ipp64f* pSrc2, int src2Stride2, Ipp64f** ppDst,
   int dstRoiShift, int dstStride2, int count);
```

Case 4: Vector array - vector array operation

```
IppStatus ippmCrossProduct_vava_32f(const Ipp32f* pSrc1, int src1Stride0,
   int src1Stride2, const Ipp32f* pSrc2, int src2Stride0, int src2Stride2,
   Ipp32f* pDst, int dstStride0, int dstStride2, int count);

IppStatus ippmCrossProduct_vava_64f(const Ipp64f* pSrc1, int src1Stride0,
   int src1Stride2, const Ipp64f* pSrc2, int src2Stride0, int src2Stride2,
   Ipp64f* pDst, int dstStride0, int dstStride2, int count);
```

```
IppStatus ippmCrossProduct_vava_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp32f** pDst, int dstRoiShift, int dstStride0, int count);
IppStatus ippmCrossProduct_vava_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp64f** ppDst, int dstRoiShift, int dstStride0, int count);
IppStatus ippmCrossProduct_vava_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2,
    Ipp32f** ppDst, int dstRoiShift, int dstStride2, int count);
IppStatus ippmCrossProduct_vava_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2,
    Ipp64f** ppDst, int dstRoiShift, int dstStride2, int count);
```

Parameters

pSrc1, ppSrc1	Pointer to the first source vector or vector array.
src1Stride0	Stride between the vectors in the first source vector array.
src1Stride2	Stride between the elements in the first source vector(s).
src1RoiShift	ROI shift in the first source vector.
pSrc2, ppSrc2	Pointer to the second source vector or vector array.
src2Stride0	Stride between the vectors in the second source vector array.
src2Stride2	Stride between the elements in the second source vector(s).
src2RoiShift	ROI shift in the second source vector(s).
pDst, ppDst	Pointer to the destination vector or vector array.
dstStride0	Stride between the vectors in the destination array.
dstStride2	Stride between the elements in the destination vector.
dstRoiShift	ROI shift in the destination vector.
count	Number of vectors in the array.

Description

The function `ippmCrossProduct` is declared in the `ippm.h` header file. The function composes a vector product of two source vectors. The first element in the destination vector is obtained by subtracting the product of the third element in the first source vector and the second element in the second source vector from the product of the second element in the first source vector and the third element in the second source vector.

The following relations are used in computing the first destination element and the other two elements:

$dst[0] = src1[1] \times src2[2] - src1[2] \times src2[1]$

$dst[1] = src1[2] \times src2[0] - src1[0] \times src2[2]$

$dst[2] = src1[0] \times src2[1] - src1[1] \times src2[0]$.

The result is stored in the destination vector.

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the input size parameter is equal to 0.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not divisible by size of data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the roiShift value is negative or not divisible by size of data type.
`ippStsCountMatrixErr`	Returns an error when the count value is less or equal to zero.

NOTE. The function `ippmCrossProduct` is only defined for three-dimensional vectors and fails with all other argument types.

DotProduct

Computes dot product of two vectors.

Syntax

Case 1: Vector - vector operation

```
IppStatus ippmDotProduct_vv_32f(const Ipp32f* pSrc1, int src1Stride2,
   const Ipp32f* pSrc2, int src2Stride2, Ipp32f* pDst, int len);
IppStatus ippmDotProduct_vv_64f(const Ipp64f* pSrc1, int src1Stride2,
   const Ipp64f* pSrc2, int src2Stride2, Ipp64f* pDst, int len);
```

```
IppStatus ippmDotProduct_vv_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
   const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f* pDst, int len);

IppStatus ippmDotProduct_vv_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
   const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f* pDst, int len);
```

Case 2: Vector array - vector operation

```
IppStatus ippmDotProduct_vav_32f(const Ipp32f* pSrc1, int src1Stride0,
   int src1Stride2, const Ipp32f* pSrc2, int src2Stride2, Ipp32f* pDst,
   int len, int count);

IppStatus ippmDotProduct_vav_64f(const Ipp64f* pSrc1, int src1Stride0,
   int src1Stride2, const Ipp64f* pSrc2, int src2Stride2, Ipp64f* pDst,
   int len, int count);

IppStatus ippmDotProduct_vav_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
   int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f* pDst,
   int len, int count);

IppStatus ippmDotProduct_vav_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
   int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f* pDst,
   int len, int count);

IppStatus ippmDotProduct_vav_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
   int src1Stride2, const Ipp32f* pSrc2, int src2Stride2, Ipp32f* pDst,
   int len, int count);

IppStatus ippmDotProduct_vav_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
   int src1Stride2, const Ipp64f* pSrc2, int src2Stride2, Ipp64f* pDst,
   int len, int count);
```

Case 3: Vector array - vector array operation

```
IppStatus ippmDotProduct_vava_32f(const Ipp32f* pSrc1, int src1Stride0,
   int src1Stride2, const Ipp32f* pSrc2, int src2Stride0, int src2Stride2,
   Ipp32f* pDst, int len, int count);

IppStatus ippmDotProduct_vava_64f(const Ipp64f* pSrc1, int src1Stride0,
   int src1Stride2, const Ipp64f* pSrc2, int src2Stride0, int src2Stride2,
   Ipp64f* pDst, int len, int count);

IppStatus ippmDotProduct_vava_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
   int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0,
   Ipp32f* pDst, int len, int count);

IppStatus ippmDotProduct_vava_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
   int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0,
   Ipp64f* pDst, int len, int count);
```

```
IppStatus ippmDotProduct_vava_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
   int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2,
   Ipp32f* pDst, int len, int count);
IppStatus ippmDotProduct_vava_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
   int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2,
   Ipp64f* pDst, int len, int count);
```

Parameters

pSrc1, ppSrc1	Pointer to the first source vector or vector array.
src1Stride0	Stride between the vectors in the first source vector array.
src1Stride2	Stride between the elements in the first source vector(s).
src1RoiShift	ROI shift in the first source vector.
pSrc2, ppSrc2	Pointer to the second source vector or vector array.
src2Stride0	Stride between the vectors in the second source vector array.
src2Stride2	Stride between the elements in the second source vector(s).
src2RoiShift	ROI shift in the second source vector(s).
pDst	Pointer to the destination value or array of values.
len	Vector length.
count	Number of vectors in the array.

Description

The function `ippmDotProduct` is declared in the `ippm.h` header file. The function computes the inner (dot) product of two source vectors by adding together the products of their respective elements and storing the resulting value in *pDst*:

$$dst = \sum_{i} src1[i] \times src2[i],$$

$0 \le i < len$.

If the operation is performed on a vector array, the resulting values are stored sequentially in the array with the pointer *pDst*.

The following example demonstrates how to use the function `ippmDotProduct_vav_32f`. For more information, see also examples in the Getting Started chapter.

Example 4-6 ippmDotProduct_vav_32f

```
IppStatus dotProduct_vav_32f(void) {
    /* Src1 is 3 vectors with length=4 */
    Ipp32f pSrc1[3*4] = { 1,  2,  4,  8,
                          11, 22, 44, 88,
                          22, 44, 88, 176 };

    /* Src2 is vector with length=4 */
    Ipp32f pSrc2[4] = { 1, 0.5, 0.25 , 0.125 };

    /* Standard description for source vectors */
    int src1Stride2 = sizeof(Ipp32f);
    int src1Stride0 = 4*sizeof(Ipp32f);
    int src2Stride2 = sizeof(Ipp32f);

    int length = 4;
    int count  = 3;

    Ipp32f pDst[3];  /* Destination is array of values */

    IppStatus status = ippmDotProduct_vav_32f((const Ipp32f*)pSrc1,
        src1Stride0, src1Stride2, (const Ipp32f*)pSrc2, src2Stride2,
        pDst, length, count);

    printf_va_Ipp32f("Src1 is 3 vectors:", pSrc1, 4, 3, status);
    printf_va_Ipp32f("Src2 is vector:", pSrc2, 4, 1, status);
    printf_va_Ipp32f("Dst is 3 values:", pDst, 3, 1, status);

    return status;
}
```

The program above produces the following output:

```
Src1 is 3 vectors:
1.000000  2.000000  4.000000  8.000000
11.000000  22.000000  44.000000  88.000000
22.000000  44.000000  88.000000  176.000000

Src2 is vector:
1.000000  0.500000  0.250000  0.125000
```

```
Dst is 3 values:
4.000000  44.000000  88.000000
```

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the input size parameter is equal to 0.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not divisible by size of data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the roiShift value is negative or not divisible by size of data type.
`ippStsCountMatrixErr`	Returns an error when the count value is less or equal to zero.

L2Norm

Computes vector L2 norm.

Syntax

Case 1: Vector operation

```
IppStatus ippmL2Norm_v_32f(const Ipp32f* pSrc, int srcStride2, Ipp32f* pDst,
    int len);
IppStatus ippmL2Norm_v_64f(const Ipp64f* pSrs, int srcStride2, Ipp64f* pDst,
    int len);
IppStatus ippmL2Norm_v_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    Ipp32f* pDst, int len);
IppStatus ippmL2Norm_v_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    Ipp64f* pDst, int len);
```

Case 2: Vector array operation

```
IppStatus ippmL2Norm_va_32f(const Ipp32f* pSrc, int srcStride0, int srcStride2,
    Ipp32f* pDst, int len, int count);
IppStatus ippmL2Norm_va_64f(const Ipp64f* pSrc, int srcStride0, int srcStride2,
    Ipp64f* pDst, int len, int count);
```

```
IppStatus ippmL2Norm_va_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp32f* pDst, int len, int count);
IppStatus ippmL2Norm_va_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp64f* pDst, int len, int count);
IppStatus ippmL2Norm_va_32f_L(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride2, Ipp32f* pDst, int len, int count);
IppStatus ippmL2Norm_va_64f_L(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride2, Ipp64f* pDst, int len, int count);
```

Parameters

pSrc, ppSrc	Pointer to the source vector or vector array.
srcStride0	Stride between the vectors in the source vector array.
srcStride2	Stride between the elements in the source vector(s).
srcRoiShift	ROI shift in the source vector(s).
pDst	Pointer to the destination value or array of values.
len	Vector length.
count	Number of vectors in the array.

Description

The function `ippmL2Norm` is declared in the `ippm.h` header file. The function calculates L2 norm of the source vector and stores the result in *pDst*. The destination value is the square root of the sum of all the squared elements in the source vector, or

$$dst = \sqrt{\sum_i src[i]^2}$$

$0 \le i < len$.

The following example demonstrates how to use the function `ippmL2Norm_va_32f_P`. For more information, see also examples in the Getting Started chapter.

Example 4-7 ippmL2Norm_va_32f_P

```
IppStatus l2norm_va_32f_P(void) {
    /* Source data */
    Ipp32f src[3*4] = { 1.1f, 2.1f, 3.1f, 4.1f,
                        1.2f, 2.2f, 3.2f, 4.2f,
                        1.3f, 2.3f, 3.3f, 4.3f };

    /*
    // The first operand is 4 vector-columns with length=3
    // Pointer description:
    // ppSrc1[0] pointer to the first element of the first column
    // ppSrc1[1] pointer to the second element of the first column
    // ppSrc1[2] pointer to the third element  of the first column
    */

    Ipp32f* ppSrc[3] = { src, src+4, src+8 };
    int srcRoiShift = 0;
    int srcStride0 = sizeof(Ipp32f); /* Stride between columns */
    int length = 3;

    /*
    // Destination is 4 values
    */
    Ipp32f pDst[4];
    int count = 4;

    IppStatus status = ippmL2Norm_va_32f_P((const Ipp32f**)ppSrc,
       srcRoiShift, srcStride0, pDst, length, count);

    printf_va_Ipp32f("Dst is 4 values:", pDst, 4, 1, status);

    return status;
}
```

The program above produces the following output:

```
Dst is 4 values:
2.083267  3.813135  5.544366  7.275988
```

Return Values

ippStsOk	Returns no error.
ippStsNullPtrErr	Returns an error when at least one input pointer is NULL.
ippStsSizeErr	Returns an error when the input size parameter is equal to 0.
ippStsStrideMatrixErr	Returns an error when the stride value is not positive or not divisible by size of data type.
ippStsRoiShiftMatrixErr	Returns an error when the roiShift value is negative or not divisible by size of data type.
ippStsCountMatrixErr	Returns an error when the count value is less or equal to zero.

LComb

Composes linear combination of two vectors.

Syntax

Case 1: Vector - vector operation

```
IppStatus ippmLComb_vv_32f(const Ipp32f* pSrc1, int src1Stride2, Ipp32f scale1,
    const Ipp32f* pSrc2, int src2Stride2, Ipp32f scale2, Ipp32f* pDst,
    int dstStride2, int len);
IppStatus ippmLComb_vv_64f(const Ipp64f* pSrc1, int src1Stride2, Ipp64f scale1,
    const Ipp64f* pSrc2, int src2Stride2, Ipp64f scale2, Ipp64f* pDst,
    int dstStride2, int len);
IppStatus ippmLComb_vv_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    Ipp32f scale1, const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f scale2,
    Ipp32f** ppDst, int dstRoiShift, int len);
IppStatus ippmLComb_vv_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    Ipp64f scale1, const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f scale2,
    Ipp64f** ppDst, int dstRoiShift, int len);
```

Case 2: Vector array - vector operation

```
IppStatus ippmLComb_vav_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride2, Ipp32f scale1, const Ipp32f* pSrc2, int src2Stride2,
    Ipp32f scale2, Ipp32f* pDst, int dstStride0, int dstStride2, int len,
    int count);
```

```
IppStatus ippmLComb_vav_64f(const Ipp64f* pSrc1, int src1Stride0,
   int src1Stride2, Ipp64f scale1, const Ipp64f* pSrc2, int src2Stride2,
   Ipp64f scale2, Ipp64f* pDst, int dstStride0, int dstStride2, int len,
   int count);

IppStatus ippmLComb_vav_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
   int src1Stride0, Ipp32f scale1, const Ipp32f** ppSrc2, int src2RoiShift,
   Ipp32f scale2, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int len,
   int count);

IppStatus ippmLComb_vav_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
   int src1Stride0, Ipp64f scale1, const Ipp64f** ppSrc2, int src2RoiShift,
   Ipp64f scale2, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int len,
   int count);

IppStatus ippmLComb_vav_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
   int src1Stride2, Ipp32f scale1, const Ipp32f* pSrc2, int src2Stride2,
   Ipp32f scale2, Ipp32f** ppDst, int dstRoiShift, int dstStride2, int len,
   int count);

IppStatus ippmLComb_vav_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
   int src1Stride2, Ipp64f scale1, const Ipp64f* pSrc2, int src2Stride2,
   Ipp64f scale2, Ipp64f** ppDst, int dstRoiShift, int dstStride2, int len,
   int count);
```

Case 3: Vector array - vector array operation

```
IppStatus ippmLComb_vava_32f(const Ipp32f* pSrc1, int src1Stride0,
   int src1Stride2, Ipp32f scale1, const Ipp32f* pSrc2, int src2Stride0,
   int src2Stride2, Ipp32f scale2, Ipp32f* pDst, int dstStride0,
   int dstStride2, int len, int count);

IppStatus ippmLComb_vava_64f(const Ipp64f* pSrc1, int src1Stride0,
   int src1Stride2, Ipp64f scale1, const Ipp64f* pSrc2, int src2Stride0,
   int src2Stride2, Ipp64f scale2, Ipp64f* pDst, int dstStride0,
   int dstStride2, int len, int count);

IppStatus ippmLComb_vava_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
   int src1Stride0, Ipp32f scale1, const Ipp32f** ppSrc2, int src2RoiShift,
   int src2Stride0, Ipp32f scale2, Ipp32f** ppDst, int dstRoiShift,
   int dstStride0, int len, int count);

IppStatus ippmLComb_vava_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
   int src1Stride0, Ipp64f scale1, const Ipp64f** ppSrc2, int src2RoiShift,
   int src2Stride0, Ipp64f scale2, Ipp64f** ppDst, int dstRoiShift,
   int dstStride0, int len, int count);
```

```
IppStatus ippmLComb_vava_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride2, Ipp32f scale1, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Stride2, Ipp32f scale2, Ipp32f** ppDst, int dstRoiShift,
    int dstStride2, int len, int count);
IppStatus ippmLComb_vava_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride2, Ipp64f scale1, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Stride2, Ipp64f scale2, Ipp64f** ppDst, int dstRoiShift,
    int dstStride2, int len, int count);
```

Parameters

pSrc1, ppSrc1	Pointer to the first source vector or vector array.
src1Stride0	Stride between the vectors in the first source vector array.
src1Stride2	Stride between the elements in the first source vector(s).
src1RoiShift	ROI shift in the first source vector.
pSrc2, ppSrc2	Pointer to the second source vector or vector array.
src2Stride0	Stride between the vectors in the second source vector array.
src2Stride2	Stride between the elements in the second source vector(s).
src2RoiShift	ROI shift in the second source vector(s).
pDst, ppDst	Pointer to the destination vector or vector array.
dstStride0	Stride between the vectors in the destination array.
dstStride2	Stride between the elements in the destination vector.
dstRoiShift	ROI shift in the destination vector.
scale1	First multiplier.
scale2	Second multiplier.
len	Vector length.
count	Number of vectors in the array.

Description

The function ippmLComb is declared in the ippm.h header file. The function composes a linear combination of two vectors by multiplying the first source vector by *scale1*, adding the result to the second source vector multiplied by *scale2*, and storing the resulting vector in *pDst*.

$$dst[i] = scale1 \times src1[i] + scale2 \times src2[i],$$

$0 \le i < len$.

The following example demonstrates how to use the function `ippmLComb_vava_32f`. For more information, see also examples in the Getting Started chapter.

Example 4-8 ippmLComb_vava_32f

```
IppStatus lcomb_vava_32f(void) {
    /* Src1 data: 4 vectors with length=3, Stride2=2*sizeof(Ipp32f) */
    Ipp32f pSrc1[4*6] = { 1, 0,  2,  0, 3,  0,
                          4, 0,  5,  0, 6,  0,
                          7, 0,  8,  0, 9,  0,
                          6, 0,  4,  0, 2, 0 };

    int src1Stride2 = 2*sizeof(Ipp32f);
    int src1Stride0 = 6*sizeof(Ipp32f);

   /* Src2 data: 4 vectors with length=3, Stride2=sizeof(Ipp32f) */
    Ipp32f pSrc2[4*3] = { -3, -1, -2,
                          -7, -3, -4,
                          -4, -5, -6,
                          -5, -2, -3 };

    int src2Stride2 = sizeof(Ipp32f);
    int src2Stride0 = 3*sizeof(Ipp32f);

    /* Dst is 4 vectors with length=3, Stride2=sizeof(Ipp32f) */
    Ipp32f pDst[4*3];
    int dstStride2=sizeof(Ipp32f);
    int dstStride0=3*sizeof(Ipp32f);

    Ipp32f scale1 = 0.5;
    Ipp32f scale2 = 1.5;
    int length = 3;
    int count  = 4;

    IppStatus status = ippmLComb_vava_32f((const Ipp32f*)pSrc1,
        src1Stride0, src1Stride2, scale1,
        (const Ipp32f*)pSrc2, src2Stride0, src2Stride2, scale2,
        pDst, dstStride0, dstStride2, length, count);

    printf_va_Ipp32f("4 destination vectors:", pDst, 3, 4, status);
    return status;
}
```

The program above produces the following output:

```
4 destination vectors:
-4.000000  -0.500000  -1.500000
-8.500000  -2.000000  -3.000000
-2.500000  -3.500000  -4.500000
-4.500000  -1.000000  -3.500000
```

Return Values

ippStsOk	Returns no error.
ippStsNullPtrErr	Returns an error when at least one input pointer is NULL.
ippStsSizeErr	Returns an error when the input size parameter is equal to 0.
ippStsStrideMatrixErr	Returns an error when the stride value is not positive or not divisible by size of data type.
ippStsRoiShiftMatrixErr	Returns an error when the roiShift value is negative or not divisible by size of data type.
ippStsCountMatrixErr	Returns an error when the count value is less or equal to zero.

Matrix Algebra Functions 5

This chapter describes Intel® Integrated Performance Primitives (Intel® IPP) functions for small matrices that perform matrix algebra operations.

Table 5-1 Matrix Algebra functions

Function Base Name	Operation
Transpose	Performs matrix transposition.
Invert	Computes matrix inverse.
FrobNorm	Computes matrix Frobenius norm.
Det	Computes matrix determinant.
Trace	Computes matrix trace.
Mul	Multiplies matrix by a constant, vector, or another matrix.
Add	Adds matrix to another matrix.
Sub	Subtracts matrix from another matrix.
Gaxpy	Performs the “gaxpy” operation on a matrix.

Transpose

Performs matrix transposition.

Syntax

Case 1: Matrix operation

```
IppStatus ippmTranspose_m_32f(const Ipp32f* pSrc, int srcStride1,
    int srcStride2, int width, int height, Ipp32f* pDst, int dstStride1,
    int dstStride2);
IppStatus ippmTranspose_m_64f(const Ipp64f* pSrc, int srcStride1,
    int srcStride2, int width, int height, Ipp64f* pDst, int dstStride1,
    int dstStride2);
IppStatus ippmTranspose_m_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    int width, int height, Ipp32f** ppDst, int dstRoiShift);
IppStatus ippmTranspose_m_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    int width, int height, Ipp64f** ppDst, int dstRoiShift);
```

Case 2: Matrix array operation

```
IppStatus ippmTranspose_ma_32f(const Ipp32f* pSrc, int srcStride0,
    int srcStride1, int srcStride2, int width, int height, Ipp32f* pDst,
    int dstStride0, int dstStride1, int dstStride2, int count);
IppStatus ippmTranspose_ma_64f(const Ipp64f* pSrc, int srcStride0,
    int srcStride1, int srcStride2, int width, int height, Ipp64f* pDst,
    int dstStride0, int dstStride1, int dstStride2, int count);
IppStatus ippmTranspose_ma_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride0, int width, int height, Ipp32f** ppDst, int dstRoiShift,
    int dstStride0, int count);
IppStatus ippmTranspose_ma_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride0, int width, int height, Ipp64f** ppDst, int dstRoiShift,
    int dstStride0, int count);
IppStatus ippmTranspose_ma_32f_L(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, int width, int height, Ipp32f** ppDst,
    int dstRoiShift, int dstStride1, int dstStride2, int count);
IppStatus ippmTranspose_ma_64f_L(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, int width, int height, Ipp64f** ppDst,
    int dstRoiShift, int dstStride1, int dstStride2, int count);
```

Parameters

pSrc, ppSrc	Pointer to the source matrix or array of matrices.
srcStride0	Stride between the matrices in the source array.
srcStride1	Stride between the rows in the source matrix(ces).
srcStride2	Stride between the elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).
width	Source matrix width.
height	Source matrix height.
pDst, ppDst	Pointer to the destination matrix or array of matrices.
dstStride0	Stride between the matrices in the destination array.
dstStride1	Stride between the rows in the destination matrix.
dstStride2	Stride between the elements in the destination matrix.
dstRoiShift	ROI shift in the destination matrix.
count	Number of matrices in the array.

Description

The function `ippmTranspose` is declared in the `ippm.h` header file. The function transposes the source matrix and stores the result in *pDst* or *ppDst*. The destination is obtained from the source matrix by transforming the columns of the source to the rows in the destination for each matrix element:

$dst[j][i] = src[i][j]$,

$0 \le i < height$, $0 \le j < width$.

Note that the destination matrix must have the number of columns equal to *height* and the number of rows equal to *width*.

The following examples demonstrate how to use the functions `ippmTranspose_m_32f`, `ippmTranspose_m_32f_P`, and `ippmTranspose_ma_32f_L`. For more information, see also examples in the Getting Started chapter.

Example 5-1 ippmTranspose_m_32f

```
IppStatus transpose_m_32f(void) {
    /* Source matrix with width=4 and height=3 */
    Ipp32f pSrc[3*4] = { 1,  2, 3, 4,
                         5,  6, 7, 8,
                         9,  0, 1, 2 };

    /* Destination matrix with width=3 and height=4 */
    Ipp32f pDst[4*3];
    /* Standard description for source and destination matrices */
    int srcStride2 = sizeof(Ipp32f);
    int srcStride1 = 4*sizeof(Ipp32f);
    int dstStride2 = sizeof(Ipp32f);
    int dstStride1 = 3*sizeof(Ipp32f);

    IppStatus status = ippmTranspose_m_32f((const Ipp32f*)pSrc,
        srcStride1, srcStride2, 4, 3, pDst, dstStride1, dstStride2);

    printf_m_Ipp32f("Transposed matrix:", pDst, 3, 4, status);
    return status;
}
```

The program above produces the following output:

```
Transposed matrix:
1.000000  5.000000  9.000000
2.000000  6.000000  0.000000
3.000000  7.000000  1.000000
4.000000  8.000000  2.000000
```

Example 5-2 ippmTranspose_m_32f_P

```
IppStatus transpose_m_32f_P(void) {
    /* Source data */
    Ipp32f src[2*6] = { 1, 0, 0, 2, 0, 3,
                        0, 4, 5, 0, 0, 6 };
    /*
    // Interested nonzero elements are refered by mask using
    // Poiner descriptor:
    // Src width=3, height=2
    */
```

Example 5-2 ippmTranspose_m_32f_P

```
    int srcRoiShift = 0;
    Ipp32f* ppSrc[2*3] = { src,   src+3, src+5,
                           src+7, src+8, src+11 };

    /*
    // Pointer description for destination matrix:
    // Dst width=2, height=3
    */

    Ipp32f  dst[3*2];
    int dstRoiShift = 0;
    Ipp32f* ppDst[3*2] = { dst,   dst+1,
                           dst+2, dst+3,
                           dst+4, dst+5 };

    IppStatus status = ippmTranspose_m_32f_P((const Ipp32f**)ppSrc,
        srcRoiShift, 3, 2, ppDst, dstRoiShift);

    printf_m_Ipp32f_P("Transposed matrix:", ppDst, 2, 3, status);
    return status;
}
```

The program above produces the following output:

```
Transposed matrix:
1.000000 4.000000
2.000000 5.000000
3.000000 6.000000
```

Example 5-3 ippmTranspose_ma_32f_L

```
IppStatus transpose_ma_32f_L(void) {
    /* Source data:
    // 3 matrices with width=4 and height=2
    */
    Ipp32f src_a[2*4] = { 10, 11, 12, 13, 24, 25, 26, 27 };
    Ipp32f src_b[2*4] = { 30, 31, 32, 33, 44, 45, 46, 47 };
    Ipp32f src_c[2*4] = { 50, 51, 52, 53, 64, 65, 66, 67 };
    /*
    // Layout description for 3 source matrices
    */
    int srcRoiShift = 0;
    int srcStride2  = sizeof(Ipp32f);
    int srcStride1  = 4*sizeof(Ipp32f);

    Ipp32f* ppSrc[3] = { src_a, src_b, src_c };
    /*
    // Layout description for destination matrices:
    // width=2, height=4, count=3
    */
    Ipp32f dst_a[4*2];
    Ipp32f dst_b[4*2];
    Ipp32f dst_c[4*2];

    Ipp32f* ppDst[3] = { dst_a, dst_b, dst_c };

    int dstRoiShift = 0;
    int dstStride2  = sizeof(Ipp32f);
    int dstStride1  = 2*sizeof(Ipp32f);

    IppStatus status = ippmTranspose_ma_32f_L((const Ipp32f**)ppSrc,
        srcRoiShift, srcStride1, srcStride2, 4, 2,
        ppDst, dstRoiShift, dstStride1, dstStride2, 3);

    printf_ma_Ipp32f_L("3 transposed matrices:", ppDst,2,4,3, status);
    return status;
}
```

The program above produces the following output:

```
3 transposed matrices:
10.000000 24.000000     30.000000 44.000000     50.000000 64.000000
11.000000 25.000000     31.000000 45.000000     51.000000 65.000000
12.000000 26.000000     32.000000 46.000000     52.000000 66.000000
13.000000 27.000000     33.000000 47.000000     53.000000 67.000000
```

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the input size parameter is equal to 0.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not dividend to size of data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the roiShift value is negative or not dividend to size of data type.
`ippStsCountMatrixErr`	Returns an error when the count value is less or equal to zero.

Invert

Computes matrix inverse.

Syntax

Case 1: Matrix operation

```
IppStatus ippmInvert_m_32f(const Ipp32f* pSrc, int srcStride1, int srcStride2,
    Ipp32f* pBuffer, Ipp32f* pDst, int dstStride1, int dstStride2,
    int widthHeight);
IppStatus ippmInvert_m_64f(const Ipp64f* pSrc, int srcStride1, int srcStride2,
    Ipp64f* pBuffer, Ipp64f* pDst, int dstStride1, int dstStride2,
    int widthHeight);
IppStatus ippmInvert_m_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    Ipp32f* pBuffer, Ipp32f** ppDst, int dstRoiShift, int widthHeight);
```

```
IppStatus ippmInvert_m_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    Ipp64f* pBuffer, Ipp64f** ppDst, int dstRoiShift, int widthHeight);
```

Case 2: Matrix array operation

```
IppStatus ippmInvert_ma_32f(const Ipp32f* pSrc, int srcStride0, int srcStride1,
    int srcStride2, Ipp32f* pBuffer, Ipp32f* pDst, int dstStride0,
    int dstStride1, int dstStride2, int widthHeight, int count);
IppStatus ippmInvert_ma_64f(const Ipp64f* pSrc, int srcStride0, int srcStride1,
    int srcStride2, Ipp64f* pBuffer, Ipp64f* pDst, int dstStride0,
    int dstStride1, int dstStride2, int widthHeight, int count);
IppStatus ippmInvert_ma_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp32f* pBuffer, Ipp32f** ppDst, int dstRoiShift,
    int dstStride0, int widthHeight, int count);
IppStatus ippmInvert_ma_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp64f* pBuffer, Ipp64f** ppDst, int dstRoiShift,
    int dstStride0, int widthHeight, int count);
IppStatus ippmInvert_ma_32f_L(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, Ipp32f* pBuffer, Ipp32f** ppDst,
    int dstRoiShift, int dstStride1, int dstStride2, int widthHeight,
    int count);
IppStatus ippmInvert_ma_64f_L(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, Ipp64f* pBuffer, Ipp64f** ppDst,
    int dstRoiShift, int dstStride1, int dstStride2, int widthHeight,
    int count);
```

Parameters

pSrc, ppSrc	Pointer to the source matrix or array of matrices.
srcStride0	Stride between the matrices in the source array.
srcStride1	Stride between the rows in the source matrix(ces).
srcStride2	Stride between the elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).
pDst, ppDst	Pointer to the destination matrix or array of matrices.
dstStride0	Stride between the matrices in the destination array.
dstStride1	Stride between the rows in the destination matrix.
dstStride2	Stride between the elements in the destination matrix.
dstRoiShift	ROI shift in the destination matrix.

widthHeight	Size of the square matrix.
pBuffer	Pointer to a pre-allocated buffer to be used for internal computations. Size of the buffer must be at least $widthHeight^2+widthHeight$.
count	Number of matrices in the array.

Description

The function ippmInvert is declared in the ippm.h header file. The function finds the inverse of the source matrix and stores the result in *pDst* or *ppDst*. Upon completion,

$dst = src^{-1}$.

The following example demonstrates how to use the function ippmInvert_m_32f. All parameter settings and descriptor types are similar to those used in ippmTransform, so for more information, see also examples for the ippmTransform function. For information on other data layout cases, see the ippmTranspose function.

Example 5-4 ippmInvert_m_32f

```
IppStatus invert_m_32f(void) {
   /* Source matrix with widthHeight=3 */
    Ipp32f pSrc[3*3] = { 1,  1, -1,
                        -1,  0,  2,
                        -1, -1,  2 };
    /*
    // Standard description for source and destination matrices
    */
    int widthHeight = 3;
    int srcStride2 = sizeof(Ipp32f);
    int srcStride1 = 3*sizeof(Ipp32f);

    Ipp32f pDst[3*3];
    int dstStride2 = sizeof(Ipp32f);
    int dstStride1 = 3*sizeof(Ipp32f);

    Ipp32f pBuffer[3*3+3]; /* Buffer location */

    IppStatus status = ippmInvert_m_32f((const Ipp32f*)pSrc, srcStride1,
        srcStride2, pBuffer, pDst, dstStride1, dstStride2, 3);
```

Example 5-4 ippmInvert_m_32f (continued)

```
    printf_m_Ipp32f("Invert matrix:", pDst, 3, 3, status);
    return status;

}
```

The program above produces the following output:

```
Invert matrix:
2.000000  -1.000000  2.000000
0.000000  1.000000  -1.000000
1.000000  0.000000  1.000000
```

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the input size parameter is equal to 0.
`ippStsSingularErr`	Returns an error when the source matrix is singular.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not dividend to size of data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the roiShift value is negative or not dividend to size of data type.
`ippStsCountMatrixErr`	Returns an error when the count value is less or equal to zero.

FrobNorm

Computes matrix Frobenius.

Syntax

Case 1: Matrix operation

```
IppStatus ippmFrobNorm_m_32f(const Ipp32f* pSrc, int srcStride1,
    int srcStride2, int width, int height, Ipp32f* pDst);
```

```
IppStatus ippmFrobNorm_m_64f(const Ipp64f* pSrc, int srcStride1,
   int srcStride2, int width, int height, Ipp64f* pDst);
IppStatus ippmFrobNorm_m_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
   int width, int height, Ipp32f* pDst);
IppStatus ippmFrobNorm_m_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
   int width, int height, Ipp64f* pDst);
```

Case 2: Matrix array operation

```
IppStatus ippmFrobNorm_ma_32f(const Ipp32f* pSrc, int srcStride0,
   int srcStride1, int srcStride2, int width, int height, Ipp32f* pDst,
   int count);
IppStatus ippmFrobNorm_ma_64f(const Ipp64f* pSrc, int srcStride0,
   int srcStride1, int srcStride2, int width, int height, Ipp64f* pDst,
   int count);
IppStatus ippmFrobNorm_ma_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
   int srcStride0, int width, int height, Ipp32f* pDst, int count);
IppStatus ippmFrobNorm_ma_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
   int srcStride0, int width, int height, Ipp64f* pDst, int count);
IppStatus ippmFrobNorm_ma_32f_L(const Ipp32f** ppSrc, int srcRoiShift,
   int srcStride1, int srcStride2, int width, int height, Ipp32f* pDst,
   int count);
IppStatus ippmFrobNorm_ma_64f_L(const Ipp64f** ppSrc, int srcRoiShift,
   int srcStride1, int srcStride2, int width, int height, Ipp64f* pDst,
   int count);
```

Parameters

pSrc, ppSrc	Pointer to the source matrix or array of matrices.
srcStride0	Stride between the matrices in the source array.
srcStride1	Stride between the rows in the source matrix(ces).
srcStride2	Stride between the elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).
pDst	Pointer to the destination value or array of values.
width	Matrix width.
height	Matrix height.
count	Number of matrices in the array.

Description

The function ippmFrobNorm is declared in the ippm.h header file. The function calculates Frobenius norm of the source matrix and stores the result in *pDst*. The destination value is the square root of the sum of all the squared elements in the source matrix, or

$$dst = \sqrt{\sum_{i,j} src[i][j]^2},$$

$0 \le i < height$, $0 \le j < width$.

If the operation is performed on a vector array, the resulting values are stored sequentially in the array with the pointer *pDst*.

The following example demonstrates how to use the function ippmFrobNorm_ma_32f_L. For more information, see also examples in the Getting Started chapter.

Example 5-5 ippmFrobNorm_ma_32f_L

```
IppStatus frobnorm_ma_32f_L(void) {
   /*  Source data  */
    Ipp32f a[2*3] = { 1.0f, 1.1f, 1.2f, 1.3f, 1.4f, 1.5f };
    Ipp32f b[2*3] = { 2.0f, 2.1f, 2.2f, 2.3f, 2.4f, 2.5f };
    Ipp32f c[2*3] = { 3.0f, 3.1f, 3.2f, 3.3f, 3.4f, 3.5f };
    Ipp32f d[2*3] = { 4.0f, 4.1f, 4.2f, 4.3f, 4.4f, 4.5f };
    /*
    // Layout description for 4 source matrices:
    */
    int width  = 3;
    int height = 2;
    int count  = 4;
    Ipp32f* ppSrc[4] = { a, b, c, d };
    int srcRoiShift = 0;
    int srcStride2 = sizeof(Ipp32f);
    int srcStride1 = 3*sizeof(Ipp32f);

    Ipp32f pDst[4];  /* Destination is array of values */

    IppStatus status = ippmFrobNorm_ma_32f_L((const Ipp32f**)ppSrc,
       srcRoiShift, srcStride1, srcStride2, width, height, pDst, count);

    printf_va_Ipp32f("Dst is 3 values:", pDst, 3, 1, status);
    return status;
}
```

The program above produces the following output:

```
Dst is 3 values:
3.090307  5.527205  7.971826
```

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the input size parameter is equal to 0.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not dividend to size of data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the roiShift value is negative or not dividend to size of data type.
`ippStsCountMatrixErr`	Returns an error when the count value is less or equal to zero.

Det

Computes matrix determinant.

Syntax

Case 1: Matrix operation

```
IppStatus ippmDet_m_32f(const Ipp32f* pSrc, int srcStride1, int srcStride2,
   int widthHeight, Ipp32f* pBuffer, Ipp32f* pDst);
IppStatus ippmDet_m_64f(const Ipp64f* pSrc, int srcStride1, int srcStride2,
   int widthHeight, Ipp64f* pBuffer, Ipp64f* pDst);
IppStatus ippmDet_m_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
   int widthHeight, Ipp32f* pBuffer, Ipp32f* pDst);
IppStatus ippmDet_m_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
   int widthHeight, Ipp64f* pBuffer, Ipp64f* pDst);
```

Case 2: Matrix array operation

```
IppStatus ippmDet_ma_32f(const Ipp32f* pSrc, int srcStride0, int srcStride1,
   int srcStride2, int widthHeight, Ipp32f* pBuffer, Ipp32f* pDst, int count);
```

```
IppStatus ippmDet_ma_64f(const Ipp64f* pSrc, int srcStride0, int srcStride1,
    int srcStride2, int widthHeight, Ipp64f* pBuffer, Ipp64f* pDst, int count);
IppStatus ippmDet_ma_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride0, int widthHeight, Ipp32f* pBuffer, Ipp32f* pDst, int count);
IppStatus ippmDet_ma_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride0, int widthHeight, Ipp64f* pBuffer, Ipp64f* pDst, int count);
IppStatus ippmDet_ma_32f_L(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, int widthHeight, Ipp32f* pBuffer,
    Ipp32f* pDst, int count);
IppStatus ippmDet_ma_64f_L(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, int widthHeight, Ipp64f* pBuffer,
    Ipp64f* pDst, int count);
```

Parameters

pSrc, ppSrc	Pointer to the source matrix or array of matrices.
srcStride0	Stride between the matrices in the source array.
srcStride1	Stride between the rows in the source matrix(ces).
srcStride2	Stride between the elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).
pDst	Pointer to the destination value or array of values.
widthHeight	Size of the square matrix.
pBuffer	Pointer to a pre-allocated buffer to be used for internal computations. Size of the buffer must be at least $widthHeight^2+widthHeight$.
count	Number of matrices in the array.

Description

The function `ippmDet` is declared in the `ippm.h` header file. The function calculates the determinant of the source matrix and stores the result in *pDst*. Upon completion,

dst = *det*(*src*).

If the operation is performed on a vector array, the resulting values are stored sequentially in the array with the pointer *pDst*.

The following example demonstrates how to use the function `ippmDet_ma_32f`. For more information, see also examples in the Getting Started chapter.

Example 5-6 ippmDet_ma_32f

```
IppStatus det_ma_32f(void) {
    /* Source data: 2 matrices with widthHeight=3 */
    Ipp32f pSrc[8*3] = { 5.0f, 1.1f, 1.2f,
                         1.3f, 1.4f, 1.5f,
                         2.0f, 2.1f, 2.2f,
                         0.0f, 0.0f, 0.0f,
                         6.3f, 2.4f, 2.5f,
                         3.0f, 3.1f, 3.2f,
                         4.3f, 4.4f, 4.5f,
                         0.0f, 0.0f, 0.0f };
    /* Standard description for 2 source matrices */
    int srcStride2 = sizeof(Ipp32f);
    int srcStride1 = 3*sizeof(Ipp32f);
    int srcStride0 = 4*3*sizeof(Ipp32f);

    int widthHeight = 3;
    int count = 2;

    Ipp32f pDst[2];         /* Destination is array of values */
    Ipp32f pBuffer[3*3+3];  /* Buffer location */

    IppStatus status = ippmDet_ma_32f((const Ipp32f*)pSrc, srcStride0,
        srcStride1, srcStride2, widthHeight, pBuffer, pDst, count);
    printf_va_Ipp32f("Dst is 2 values:", pDst, 2, 1, status);
    return status;
}
```

The program above produces the following output:

```
Dst is 2 values:
-0.279999  -0.520004
```

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the input size parameter is equal to 0.

ippStsStrideMatrixErr	Returns an error when the stride value is not positive or not dividend to size of data type.
ippStsRoiShiftMatrixErr	Returns an error when the roiShift value is negative or not dividend to size of data type.
ippStsCountMatrixErr	Returns an error when the count value is less or equal to zero.

Trace

Computes matrix trace.

Syntax

Case 1: Matrix operation

```
IppStatus ippmTrace_m_32f(const Ipp32f* pSrc, int srcStride1, int srcStride2,
    int widthHeight, Ipp32f* pDst);
IppStatus ippmTrace_m_64f(const Ipp64f* pSrc, int srcStride1, int srcStride2,
    int widthHeight, Ipp64f* pDst);
IppStatus ippmTrace_m_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    int widthHeight, Ipp32f* pDst);
IppStatus ippmTrace_m_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    int widthHeight, Ipp64f* pDst);
```

Case 2: Matrix array operation

```
IppStatus ippmTrace_ma_32f(const Ipp32f* pSrc, int srcStride0, int srcStride1,
    int srcStride2, int widthHeight, Ipp32f* pDst, int count);
IppStatus ippmTrace_ma_64f(const Ipp64f* pSrc, int srcStride0, int srcStride1,
    int srcStride2, int widthHeight, Ipp64f* pDst, int count);
IppStatus ippmTrace_ma_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride0, int widthHeight, Ipp32f* pDst, int count);
IppStatus ippmTrace_ma_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride0, int widthHeight, Ipp64f* pDst, int count);
IppStatus ippmTrace_ma_32f_L(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, int widthHeight, Ipp32f* pDst, int count);
IppStatus ippmTrace_ma_64f_L(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, int widthHeight, Ipp64f* pDst, int count);
```

Parameters

pSrc, ppSrc	Pointer to the source matrix or array of matrices.
srcStride0	Stride between the matrices in the source array.
srcStride1	Stride between the rows in the source matrix(ces).
srcStride2	Stride between the elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).
pDst	Pointer to the destination value or array of values.
widthHeight	Size of the square matrix.
count	Number of matrices in the array.

Description

The function `ippmTrace` is declared in the `ippm.h` header file. The function sums up the elements along the principal diagonal of the source matrix and stores the result in *pDst*. The following formula illustrates how trace is calculated:

$$dst = \sum_{i} src[i][i],$$

$0 \le i < widthHeight$.

If the operation is performed on a vector array, the resulting values are stored sequentially in the array with the pointer *pDst*.

The following example demonstrates how to use the function `ippmTrace_ma_32f_P`. For more information, see also examples in the Getting Started chapter.

Example 5-7 ippmTrace_ma_32f_P

```
IppStatus trace_ma_32f_P(void) {
   /*
    // Source data:
    // 2 matrices with widthHeight=3
    */
```

Example 5-7 ippmTrace_ma_32f_P (continued)

```
    Ipp32f src[6*6] = { 1, 0, 0, 1, 0, 1,
                        0, 1, 1, 0, 0, 1,
                        0, 0, 1, 1, 0, 1,
                        2, 0, 0, 2, 0, 2,
                        0, 2, 2, 0, 0, 2,
                        0, 0, 2, 2, 0, 2};
    /*
    // Interested nonzero elements are refered by mask using
    // pointer descriptor
    */
    int widthHeight=3;
    int count=2;
    int srcRoiShift= 0;
    int srcStride0 = 3*6*sizeof(Ipp32f);

    Ipp32f* ppSrc[3*3] = { src,    src+3,  src+5,
                           src+7,  src+8,  src+11,
                           src+14, src+15, src+17 };

    Ipp32f pDst[2]; /* Destination is array of values */

    IppStatus status = ippmTrace_ma_32f_P((const Ipp32f**)ppSrc,
        srcRoiShift, srcStride0, widthHeight, pDst, 2);

    printf_va_Ipp32f("Dst is 2 values:", pDst, 2, 1, status);
    return status;
}
```

The program above produces the following output:

```
Dst is 2 values:
3.000000  6.000000
```

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the input size parameter is equal to 0.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not dividend to size of data type.

`ippStsRoiShiftMatrixErr`	Returns an error when a roiShift value is negative or not dividend to size of data type.
`ippStsCountMatrixErr`	Returns an error when a count value is less or equal to zero.

Mul

Multiplies matrix by a constant, a vector or another matrix.

Syntax

Case 1: Matrix - constant operation

```
IppStatus ippmMul_mc_32f(const Ipp32f* pSrc, int srcStride1, int srcStride2,
    Ipp32f val, Ipp32f* pDst, int dstStride1, int dstStride2, int width,
    int height);
IppStatus ippmMul_mc_64f(const Ipp64f* pSrc, int srcStride1, int srcStride2,
    Ipp64f val, Ipp64f* pDst, int dstStride1, int dstStride2, int width,
    int height);
IppStatus ippmMul_mc_32f_P(const Ipp32f** ppSrc, int srcRoiShift, Ipp32f val,
    Ipp32f** ppDst, int dstRoiShift, int width, int height);
IppStatus ippmMul_mc_64f_P(const Ipp64f** ppSrc, int srcRoiShift, Ipp64f val,
    Ipp64f** ppDst, int dstRoiShift, int width, int height);
```

Case 2: Transposed matrix - constant operation

```
IppStatus ippmMul_tc_32f(const Ipp32f* pSrc, int srcStride1, int srcStride2,
    Ipp32f val, Ipp32f* pDst, int dstStride1, int dstStride2, int width,
    int height);
IppStatus ippmMul_tc_64f(const Ipp64f* pSrc, int srcStride1, int srcStride2,
    Ipp64f val, Ipp64f* pDst, int dstStride1, int dstStride2, int width,
    int height);
IppStatus ippmMul_tc_32f_P(const Ipp32f** ppSrc, int srcRoiShift, Ipp32f val,
    Ipp32f** ppDst, int dstRoiShift, int width, int height);
IppStatus ippmMul_tc_64f_P(const Ipp64f** ppSrc, int srcRoiShift, Ipp64f val,
    Ipp64f** pDst, int dstRoiShift, int width, int height);
```

Case 3: Matrix array - constant operation

```
IppStatus ippmMul_mac_32f(const Ipp32f* pSrc, int srcStride0, int srcStride1,
    int srcStride2, Ipp32f val, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmMul_mac_64f(const Ipp64f* pSrc, int srcStride0, int srcStride1,
    int srcStride2, Ipp64f val, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmMul_mac_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp32f val, Ipp32f** ppDst, int dstRoiShift, int dstStride0,
    int width, int height, int count);

IppStatus ippmMul_mac_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp64f val, Ipp64f** ppDst, int dstRoiShift, int dstStride0,
    int width, int height, int count);

IppStatus ippmMul_mac_32f_L(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, Ipp32f val, Ipp32f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmMul_mac_64f_L(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, Ipp64f val, Ipp64f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int width, int height, int count);
```

Case 4: Transposed matrix array - constant operation

```
IppStatus ippmMul_tac_32f(const Ipp32f* pSrc, int srcStride0, int srcStride1,
    int srcStride2, Ipp32f val, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmMul_tac_64f(const Ipp64f* pSrc, int srcStride0, int srcStride1,
    int srcStride2, Ipp64f val, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmMul_tac_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp32f val, Ipp32f** ppDst, int dstRoiShift, int dstStride0,
    int width, int height, int count);

IppStatus ippmMul_tac_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp64f val, Ipp64f** ppDst, int dstRoiShift, int dstStride0,
    int width, int height, int count);

IppStatus ippmMul_tac_32f_L(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, Ipp32f val, Ipp32f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmMul_tac_64f_L(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, Ipp64f val, Ipp64f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int width, int height, int count);
```

Case 5: Matrix - vector operation

```
IppStatus ippmMul_mv_32f(const Ipp32f* pSrc1, int src1Stride1, int src1Stride2,
    int src1Width, int src1Height, const Ipp32f* pSrc2, int src2Stride2,
    int src2Len, Ipp32f* pDst, int dstStride2);

IppStatus ippmMul_mv_64f(const Ipp64f* pSrc1, int src1Stride1, int src1Stride2,
    int src1Width, int src1Height, const Ipp64f* pSrc2, int src2Stride2,
    int src2Len, Ipp64f* pDst, int dstStride2);

IppStatus ippmMul_mv_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Len, Ipp32f** ppDst, int dstRoiShift);

IppStatus ippmMul_mv_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Len, Ipp64f** ppDst, int dstRoiShift);
```

Case 6: Transposed matrix - vector operation

```
IppStatus ippmMul_tv_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp32f* pSrc2,
    int src2Stride2, int src2Len, Ipp32f* pDst, int dstStride2);

IppStatus ippmMul_tv_64f(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp64f* pSrc2,
    int src2Stride2, int src2Len, Ipp64f* pDst, int dstStride2);

IppStatus ippmMul_tv_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Len, Ipp32f** ppDst, int dstRoiShift);

IppStatus ippmMul_tv_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Len, Ipp64f** ppDst, int dstRoiShift);
```

Case 7: Matrix - vector array operation

```
IppStatus ippmMul_mva_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp32f* pSrc2,
    int src2Stride0, int src2Stride2, int src2Len, Ipp32f* pDst, int dstStride0,
    int dstStride2, int count);

IppStatus ippmMul_mva_64f(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp64f* pSrc2,
    int src2Stride0, int src2Stride2, int src2Len, Ipp64f* pDst, int dstStride0,
    int dstStride2, int count);

IppStatus ippmMul_mva_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Stride0, int src2Len, Ipp32f** ppDst, int dstRoiShift,
    int dstStride0, int count);
```

```
IppStatus ippmMul_mva_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Stride0, int src2Len, Ipp64f** ppDst, int dstRoiShift,
    int dstStride0, int count);

IppStatus ippmMul_mva_32f_L(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp32f** ppSrc2,
    int src2RoiShift, int src2Stride2, int src2Len, Ipp32f** ppDst,
    int dstRoiShift, int dstStride2, int count);

IppStatus ippmMul_mva_64f_L(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp64f** ppSrc2,
    int src2RoiShift, int src2Stride2, int src2Len, Ipp64f** ppDst,
    int dstRoiShift, int dstStride2, int count);
```

Case 8: Transposed matrix - vector array operation

```
IppStatus ippmMul_tva_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp32f* pSrc2,
    int src2Stride0, int src2Stride2, int src2Len, Ipp32f* pDst, int dstStride0,
    int dstStride2, int count);

IppStatus ippmMul_tva_64f(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp64f* pSrc2,
    int src2Stride0, int src2Stride2, int src2Len, Ipp64f* pDst, int dstStride0,
    int dstStride2, int count);

IppStatus ippmMul_tva_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Stride0, int src2Len, Ipp32f** ppDst, int dstRoiShift,
    int dstStride0, int count);

IppStatus ippmMul_tva_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Stride0, int src2Len, Ipp64f** ppDst, int dstRoiShift,
    int dstStride0, int count);

IppStatus ippmMul_tva_32f_L(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp32f** ppSrc2,
    int src2RoiShift, int src2Stride2, int src2Len, Ipp32f** ppDst,
    int dstRoiShift, int dstStride2, int count);

IppStatus ippmMul_tva_64f_L(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp64f** ppSrc2,
    int src2RoiShift, int src2Stride2, int src2Len, Ipp64f** ppDst,
    int dstRoiShift, int dstStride2, int count);
```

Case 9: Matrix array - vector operation

```
IIppStatus ippmMul_mav_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp32f* pSrc2, int src2Stride2, int src2Len, Ipp32f* pDst,
    int dstStride0, int dstStride2, int count);

IppStatus ippmMul_mav_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp64f* pSrc2, int src2Stride2, int src2Len, Ipp64f* pDst,
    int dstStride0, int dstStride2, int count);

IppStatus ippmMul_mav_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, int src1Width, int src1Height, const Ipp32f** ppSrc2,
    int src2RoiShift, int src2Len, Ipp32f** ppDst, int dstRoiShift,
    int dstStride0, int count);

IppStatus ippmMul_mav_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, int src1Width, int src1Height, const Ipp64f** ppSrc2,
    int src2RoiShift, int src2Len, Ipp64f** ppDst, int dstRoiShift,
    int dstStride0, int count);

IppStatus ippmMul_mav_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp32f* pSrc2, int src2Stride2, int src2Len, Ipp32f** ppDst,
    int dstRoiShift, int dstStride2, int count);

IppStatus ippmMul_mav_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp64f* pSrc2, int src2Stride2, int src2Len, Ipp64f** ppDst,
    int dstRoiShift, int dstStride2, int count);
```

Case 10: Transposed matrix array - vector operation

```
IppStatus ippmMul_tav_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp32f* pSrc2, int src2Stride2, int src2Len, Ipp32f* pDst,
    int dstStride0, int dstStride2, int count);

IppStatus ippmMul_tav_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp64f* pSrc2, int src2Stride2, int src2Len, Ipp64f* pDst,
    int dstStride0, int dstStride2, int count);

IppStatus ippmMul_tav_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, int src1Width, int src1Height, const Ipp32f** ppSrc2,
    int src2RoiShift, int src2Len, Ipp32f** ppDst, int dstRoiShift,
    int dstStride0, int count);
```

```
IppStatus ippmMul_tav_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, int src1Width, int src1Height, const Ipp64f** ppSrc2,
    int src2RoiShift, int src2Len, Ipp64f** ppDst, int dstRoiShift,
    int dstStride0, int count);

IppStatus ippmMul_tav_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp32f* pSrc2, int src2Stride2, int src2Len, Ipp32f** ppDst,
    int dstRoiShift, int dstStride2, int count);

IppStatus ippmMul_tav_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp64f* pSrc2, int src2Stride2, int src2Len, Ipp64f** ppDst,
    int dstRoiShift, int dstStride2, int count);
```

Case 11: Matrix array - vector array operation

```
IppStatus ippmMul_mava_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp32f* pSrc2, int src2Stride0, int src2Stride2, int src2Len,
    Ipp32f* pDst, int dstStride0, int dstStride2, int count);

IppStatus ippmMul_mava_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp64f* pSrc2, int src2Stride0, int src2Stride2, int src2Len,
    Ipp64f* pDst, int dstStride0, int dstStride2, int count);

IppStatus ippmMul_mava_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, int src1Width, int src1Height, const Ipp32f** ppSrc2,
    int src2RoiShift, int src2Stride0, int src2Len, Ipp32f** ppDst,
    int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_mava_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, int src1Width, int src1Height, const Ipp64f** ppSrc2,
    int src2RoiShift, int src2Stride0, int src2Len, Ipp64f** ppDst,
    int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_mava_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2, int src2Len,
    Ipp32f** ppDst, int dstRoiShift, int dstStride2, int count);

IppStatus ippmMul_mava_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2, int src2Len,
    Ipp64f** ppDst, int dstRoiShift, int dstStride2, int count);
```

Case 12: Transposed matrix array - vector array operation

```
IppStatus ippmMul_tava_32f(const Ipp32f* pSrc1, int src1Stride0,
   int src1Stride1, int src1Stride2, int src1Width, int src1Height,
   const Ipp32f* pSrc2, int src2Stride0, int src2Stride2, int src2Len,
   Ipp32f* pDst, int dstStride0, int dstStride2, int count);

IppStatus ippmMul_tava_64f(const Ipp64f* pSrc1, int src1Stride0,
   int src1Stride1, int src1Stride2, int src1Width, int src1Height,
   const Ipp64f* pSrc2, int src2Stride0, int src2Stride2, int src2Len,
   Ipp64f* pDst, int dstStride0, int dstStride2, int count);

IppStatus ippmMul_tava_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
   int src1Stride0, int src1Width, int src1Height, const Ipp32f** ppSrc2,
   int src2RoiShift, int src2Stride0, int src2Len, Ipp32f** ppDst,
   int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_tava_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
   int src1Stride0, int src1Width, int src1Height, const Ipp64f** ppSrc2,
   int src2RoiShift, int src2Stride0, int src2Len, Ipp64f** ppDst,
   int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_tava_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
   int src1Stride1, int src1Stride2, int src1Width, int src1Height,
   const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2, int src2Len,
   Ipp32f** ppDst, int dstRoiShift, int dstStride2, int count);

IppStatus ippmMul_tava_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
   int src1Stride1, int src1Stride2, int src1Width, int src1Height,
   const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2, int src2Len,
   Ipp64f** ppDst, int dstRoiShift, int dstStride2, int count);
```

Case 13: Matrix - matrix operation

```
IppStatus ippmMul_mm_32f(const Ipp32f* pSrc1, int src1Stride1, int src1Stride2,
   int src1Width, int src1Height, const Ipp32f* pSrc2, int src2Stride1,
   int src2Stride2, int src2Width, int src2Height, Ipp32f* pDst,
   int dstStride1, int dstStride2);

IppStatus ippmMul_mm_64f(const Ipp64f* pSrc1, int src1Stride1, int src1Stride2,
   int src1Width, int src1Height, const Ipp64f* pSrc2, int src2Stride1,
   int src2Stride2, int src2Width, int src2Height, Ipp64f* pDst,
   int dstStride1, int dstStride2);

IppStatus ippmMul_mm_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
   int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift,
   int src2Width, int src2Height, Ipp32f** ppDst, int dstRoiShift);

IppStatus ippmMul_mm_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
   int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift,
   int src2Width, int src2Height, Ipp64f** ppDst, int dstRoiShift);
```

Case 14: Transposed matrix - matrix operation

```
IppStatus ippmMul_tm_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp32f* pSrc2,
    int src2Stride1, int src2Stride2, int src2Width, int src2Height,
    Ipp32f* pDst, int dstStride1, int dstStride2);

IppStatus ippmMul_tm_64f(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp64f* pSrc2,
    int src2Stride1, int src2Stride2, int src2Width, int src2Height,
    Ipp64f* pDst, int dstStride1, int dstStride2);

IppStatus ippmMul_tm_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Width, int src2Height, Ipp32f** ppDst, int dstRoiShift);

IppStatus ippmMul_tm_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Width, int src2Height, Ipp64f** ppDst, int dstRoiShift);
```

Case 15: Matrix - transposed matrix operation

```
IppStatus ippmMul_mt_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp32f* pSrc2,
    int src2Stride1, int src2Stride2, int src2Width, int src2Height,
    Ipp32f* pDst, int dstStride1, int dstStride2);

IppStatus ippmMul_mt_64f(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp64f* pSrc2,
    int src2Stride1, int src2Stride2, int src2Width, int src2Height,
    Ipp64f* pDst, int dstStride1, int dstStride2);

IppStatus ippmMul_mt_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Width, int src2Height, Ipp32f** ppDst, int dstRoiShift);

IppStatus ippmMul_mt_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Width, int src2Height, Ipp64f** ppDst, int dstRoiShift);
```

Case 16: Transposed matrix - transposed matrix operation

```
IppStatus ippmMul_tt_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp32f* pSrc2,
    int src2Stride1, int src2Stride2, int src2Width, int src2Height,
    Ipp32f* pDst, int dstStride1, int dstStride2);

IppStatus ippmMul_tt_64f(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp64f* pSrc2,
    int src2Stride1, int src2Stride2, int src2Width, int src2Height,
    Ipp64f* pDst, int dstStride1, int dstStride2);
```

```
IppStatus ippmMul_tt_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Width, int src2Height, Ipp32f** ppDst, int dstRoiShift);

IppStatus ippmMul_tt_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Width, int src2Height, Ipp64f** ppDst, int dstRoiShift);
```

Case 17: Matrix - matrix array operation

```
IppStatus ippmMul_mma_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp32f* pSrc2,
    int src2Stride0, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_mma_64f(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp64f* pSrc2,
    int src2Stride0, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_mma_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Stride0, int src2Width, int src2Height, Ipp32f** ppDst,
    int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_mma_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Stride0, int src2Width, int src2Height, Ipp64f** ppDst,
    int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_mma_32f_L(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp32f** ppSrc2,
    int src2RoiShift, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_mma_64f_L(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp64f** ppSrc2,
    int src2RoiShift, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int count);
```

Case 18: Transposed matrix - matrix array operation

```
IppStatus ippmMul_tma_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp32f* pSrc2,
    int src2Stride0, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_tma_64f(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp64f* pSrc2,
    int src2Stride0, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_tma_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Stride0, int src2Width, int src2Height, Ipp32f** ppDst,
    int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_tma_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Stride0, int src2Width, int src2Height, Ipp64f** ppDst,
    int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_tma_32f_L(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp32f** ppSrc2,
    int src2RoiShift, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_tma_64f_L(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp64f** ppSrc2,
    int src2RoiShift, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int count);
```

Case 19: Matrix - transposed matrix array operation

```
IppStatus ippmMul_mta_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp32f* pSrc2,
    int src2Stride0, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_mta_64f(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp64f* pSrc2,
    int src2Stride0, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);
```

```
IppStatus ippmMul_mta_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Stride0, int src2Width, int src2Height, Ipp32f** ppDst,
    int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_mta_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Stride0, int src2Width, int src2Height, Ipp64f** ppDst,
    int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_mta_32f_L(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp32f** ppSrc2,
    int src2RoiShift, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_mta_64f_L(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp64f** ppSrc2,
    int src2RoiShift, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int count);
```

Case 20: Transposed matrix - transposed matrix array operation

```
IppStatus ippmMul_tta_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp32f* pSrc2,
    int src2Stride0, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_tta_64f(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp64f* pSrc2,
    int src2Stride0, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_tta_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Stride0, int src2Width, int src2Height, Ipp32f** ppDst,
    int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_tta_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Stride0, int src2Width, int src2Height, Ipp64f** ppDst,
    int dstRoiShift, int dstStride0, int count);
```

```
IppStatus ippmMul_tta_32f_L(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp32f** ppSrc2,
    int src2RoiShift, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_tta_64f_L(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp64f** ppSrc2,
    int src2RoiShift, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int count);
```

Case 21: Matrix array - matrix operation

```
IppStatus ippmMul_mam_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp32f* pSrc2, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_mam_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp64f* pSrc2, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_mam_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, int src1Width, int src1Height, const Ipp32f** ppSrc2,
    int src2RoiShift, int src2Width, int src2Height, Ipp32f** ppDst,
    int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_mam_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, int src1Width, int src1Height, const Ipp64f** ppSrc2,
    int src2RoiShift, int src2Width, int src2Height, Ipp64f** ppDst,
    int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_mam_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp32f* pSrc2, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_mam_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp64f* pSrc2, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int count);
```

Case 22: Transposed matrix array - matrix operation

```
IppStatus ippmMul_tam_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp32f* pSrc2, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_tam_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp64f* pSrc2, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_tam_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, int src1Width, int src1Height, const Ipp32f** ppSrc2,
    int src2RoiShift, int src2Width, int src2Height, Ipp32f** ppDst,
    int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_tam_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, int src1Width, int src1Height, const Ipp64f** ppSrc2,
    int src2RoiShift, int src2Width, int src2Height, Ipp64f** ppDst,
    int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_tam_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp32f* pSrc2, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_tam_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp64f* pSrc2, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int count);
```

Case 23: Matrix array - transposed matrix operation

```
IppStatus ippmMul_mat_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp32f* pSrc2, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_mat_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp64f* pSrc2, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);
```

```
IppStatus ippmMul_mat_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, int src1Width, int src1Height, const Ipp32f** ppSrc2,
    int src2RoiShift, int src2Width, int src2Height, Ipp32f** ppDst,
    int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_mat_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, int src1Width, int src1Height, const Ipp64f** ppSrc2,
    int src2RoiShift, int src2Width, int src2Height, Ipp64f** ppDst,
    int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_mat_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp32f* pSrc2, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_mat_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp64f* pSrc2, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int count);
```

Case 24: Transposed matrix array - transposed matrix operation

```
IppStatus ippmMul_tat_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp32f* pSrc2, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_tat_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp64f* pSrc2, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_tat_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, int src1Width, int src1Height, const Ipp32f** ppSrc2,
    int src2RoiShift, int src2Width, int src2Height, Ipp32f** ppDst,
    int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_tat_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, int src1Width, int src1Height, const Ipp64f** ppSrc2,
    int src2RoiShift, int src2Width, int src2Height, Ipp64f** ppDst,
    int dstRoiShift, int dstStride0, int count);
```

```
IppStatus ippmMul_tat_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp32f* pSrc2, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_tat_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp64f* pSrc2, int src2Stride1, int src2Stride2, int src2Width,
    int src2Height, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int count);
```

Case 25: Matrix array - matrix array operation

```
IppStatus ippmMul_mama_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp32f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2,
    int src2Width, int src2Height, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_mama_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp64f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2,
    int src2Width, int src2Height, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_mama_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, int src1Width, int src1Height, const Ipp32f** ppSrc2,
    int src2RoiShift, int src2Stride0, int src2Width, int src2Height,
    Ipp32f** ppDst, int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_mama_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, int src1Width, int src1Height, const Ipp64f** ppSrc2,
    int src2RoiShift, int src2Stride0, int src2Width, int src2Height,
    Ipp64f** ppDst, int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_mama_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride1, int src2Stride2,
    int src2Width, int src2Height, Ipp32f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int count);

IppStatus ippmMul_mama_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride1, int src2Stride2,
    int src2Width, int src2Height, Ipp64f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int count);
```

Case 26: Transposed matrix array - matrix array operation

```
IppStatus ippmMul_tama_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp32f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2,
    int src2Width, int src2Height, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_tama_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp64f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2,
    int src2Width, int src2Height, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_tama_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, int src1Width, int src1Height, const Ipp32f** ppSrc2,
    int src2RoiShift, int src2Stride0, int src2Width, int src2Height,
    Ipp32f** ppDst, int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_tama_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, int src1Width, int src1Height, const Ipp64f** ppSrc2,
    int src2RoiShift, int src2Stride0, int src2Width, int src2Height,
    Ipp64f** ppDst, int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_tama_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride1, int src2Stride2,
    int src2Width, int src2Height, Ipp32f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int count);

IppStatus ippmMul_tama_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride1, int src2Stride2,
    int src2Width, int src2Height, Ipp64f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int count);
```

Case 27: Matrix array - transposed matrix array operation

```
IppStatus ippmMul_mata_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp32f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2,
    int src2Width, int src2Height, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_mata_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp64f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2,
    int src2Width, int src2Height, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);
```

```
IppStatus ippmMul_mata_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, int src1Width, int src1Height, const Ipp32f** ppSrc2,
    int src2RoiShift, int src2Stride0, int src2Width, int src2Height,
    Ipp32f** ppDst, int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_mata_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, int src1Width, int src1Height, const Ipp64f** ppSrc2,
    int src2RoiShift, int src2Stride0, int src2Width, int src2Height,
    Ipp64f** ppDst, int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_mata_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride1, int src2Stride2,
    int src2Width, int src2Height, Ipp32f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int count);

IppStatus ippmMul_mata_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride1, int src2Stride2,
    int src2Width, int src2Height, Ipp64f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int count);
```

Case 28: Transposed matrix array - transposed matrix array operation

```
IppStatus ippmMul_tata_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp32f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2,
    int src2Width, int src2Height, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_tata_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp64f* pSrc2, int src2Stride0, int src2Stride1, int src2Stride2,
    int src2Width, int src2Height, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int count);

IppStatus ippmMul_tata_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, int src1Width, int src1Height, const Ipp32f** ppSrc2,
    int src2RoiShift, int src2Stride0, int src2Width, int src2Height,
    Ipp32f** ppDst, int dstRoiShift, int dstStride0, int count);

IppStatus ippmMul_tata_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, int src1Width, int src1Height, const Ipp64f** ppSrc2,
    int src2RoiShift, int src2Stride0, int src2Width, int src2Height,
    Ipp64f** ppDst, int dstRoiShift, int dstStride0, int count);
```

```
IppStatus ippmMul_tata_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride1, int src2Stride2,
    int src2Width, int src2Height, Ipp32f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int count);
IppStatus ippmMul_tata_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int src1Width, int src1Height,
    const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride1, int src2Stride2,
    int src2Width, int src2Height, Ipp64f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int count);
```

Parameters

pSrc, ppSrc	Pointer to the source matrix or array of matrices.
srcStride0	Stride between the matrices in the source array.
srcStride1	Stride between the rows in the source matrix(ces).
srcStride2	Stride between the elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).
pSrc1, ppSrc1	Pointer to the first source matrix or array of matrices.
src1Stride0	Stride between the matrices in the first source array.
src1Stride1	Stride between the rows in the first source matrix(ces).
src1Stride2	Stride between the elements in the first source matrix(ces).
src1RoiShift	ROI shift in the first source matrix(ces).
src1Width	Width of the first source matrix(ces).
src1Height	Height of the first source matrix(ces).
pSrc2, ppSrc2	Pointer to the second source matrix or array of matrices.
src2Stride0	Stride between the matrices in the second source array.
src2Stride1	Stride between the rows in the second source matrix(ces).
src2Stride2	Stride between the elements in the second source matrix(ces).
src2RoiShift	ROI shift in the second source matrix(ces).
src2Width	Width of the second source matrix(ces).
src2Height	Height of the second source matrix(ces).

pDst, ppDst	Pointer to the destination matrix or array of matrices.
dstStride0	Stride between the matrices in the destination array.
dstStride1	Stride between the rows in the destination matrix.
dstStride2	Stride between the elements in the destination matrix.
dstRoiShift	ROI shift in the destination matrix.
val	Multiplier used in matrix scaling.
src2Len	Vector length.
width	Matrix width.
height	Matrix height.
count	Number of matrices in the array.

Description

The function `ippmMul` is declared in the `ippm.h` header file. Like all other Intel IPP matrix operating functions, this function is parameter sensitive. All input parameters that follow the function name immediately after the underscore determine the way in which the function performs and the arguments it takes, whether it is a constant, a vector, or another matrix. This implies that with every complete function name only some of the listed arguments appear in the input list, while others are omitted.

When performed on a constant and a matrix (cases 1, 3), the function scales the source matrix by multiplying each element of the source by *val*, and stores the result in *pDst*:

$dst[i][j] = scale \times src[i][j]$,

$0 \le i < height$, $0 \le j < width$.

The following example demonstrates how to use the function `ippmMul_mac_32f`. For more information, see also examples in the Getting Started chapter.

Example 5-8 ippmMul_mac_32f

```
IppStatus mul_mac_32f(void){
    /* Source data: 2 matrices with width=3 and height=3 */
    Ipp32f pSrc[8*3] = { 3.0f, 1.1f, 1.2f,
                         1.3f, 1.4f, 1.5f,
                         2.0f, 2.1f, 2.2f,
                         0.0f, 0.0f, 0.0f,
                         3.3f, 2.4f, 2.5f,
                         3.0f, 3.1f, 3.2f,
                         4.3f, 4.4f, 4.5f,
                         0.0f, 0.0f, 0.0f };
    /* Standard description for 2 source matrices */
    int srcStride2 = sizeof(Ipp32f);
    int srcStride1 = 3*sizeof(Ipp32f);
    int srcStride0 = 4*3*sizeof(Ipp32f);

    Ipp32f val=2.0;

    /* Standard description for 2 destination matrices */
    Ipp32f pDst[2*3*3];
    int dstStride2 = sizeof(Ipp32f);
    int dstStride1 = 3*sizeof(Ipp32f);
    int dstStride0 = 9*sizeof(Ipp32f);

    int width  = 3;
    int height = 3;
    int count  = 2;

    IppStatus status = ippmMul_mac_32f((const Ipp32f*)pSrc, srcStride0,
        srcStride1, srcStride2, val, pDst, dstStride0, dstStride1,
        dstStride2, width, height, count);

    printf_ma_Ipp32f("Destination matrices:", pDst, 3, 3, 2, status);
    return status;
}
```

The program above produces the following output:

```
Destination matrices:
6.000000 2.200000 2.400000     6.600000 4.800000 5.000000
2.600000 2.800000 3.000000     6.000000 6.200000 6.400000
4.000000 4.200000 4.400000     8.600000 8.800000 9.000000
```

When performed on a constant and a transposed matrix (cases 2, 4), the function scales the transposed source matrix by multiplying each element of the source matrix by *val* and stores the result in *pDst*:

$dst[i][j] = c \times src[j][i]$,

$0 \le i < height$, $0 \le j < width$.

Note that if the operation is performed on a transposed matrix (object type `t`) or a transposed matrix array (object type `ta`), the source matrices must have the number of columns equal to *height* and the number of rows equal to *width*.

When performed on a vector and a matrix (cases 5, 7, 9, 11), the function multiplies all elements in a row of the source matrix by the respective elements in the source vector. Done in loop through all rows in the source matrix, this operation gives the destination vector, which is stored in *pDst*. The following formula applies to all matrix rows *i* and a vector:

$$dst[i] = \sum_{j} src1[i][j] \times src2[j],$$

$0 \le i < src1Height$, $0 \le j < src1Width$.

Note that the number of elements in the source vector *src2Len* must be equal to *src1Width*.

The following example demonstrates how to use the function `ippmMul_mva_32f_L`. For more information, see also examples in the Getting Started chapter.

Example 5-9 ippmMul_mva_32f_L

```
IppStatus mul_mva_32f_L(void) {
    /* Src1 matrix with width=3, height=4, Stride2=2*sizeof(Ipp32f) */
    Ipp32f pSrc1[4*6] = { 1, 0, 2, 0, 3, 0,
                          4, 0, 5, 0, 6, 0,
                          7, 0, 8, 0, 9, 0,
                          2, 0, 4, 0, 6, 0 };
    /* Standard description for Src1 */
    int src1Width   = 3;
    int src1Height  = 4;
    int src1Stride2 = 2*sizeof(Ipp32f);
    int src1Stride1 = 6*sizeof(Ipp32f);
```

Example 5-9 ippmMul_mva_32f_L (continued)

```
    /* Src2 data: 2 vectors with length=3, Stride2=sizeof(Ipp32f) */
    Ipp32f src2_a[3] = { 1, 6, 3 };
    Ipp32f src2_b[3] = { 4, 5, 2 };

    /* Layout description for Src2 */
    Ipp32f* ppSrc2[2] = { src2_a, src2_b };
    int src2RoiShift = 0;
    int src2Stride2  = sizeof(Ipp32f);
    int src2Length = 3;
    /*
    // Destination vector has length=src1Height=4
    // Layout description for Dst:
    */
    Ipp32f dst[2*4];
    Ipp32f* ppDst[4] = { dst, dst+4 };
    int dstRoiShift  = 0;
    int dstStride2  = sizeof(Ipp32f);

    int count = 2;

    IppStatus status = ippmMul_mva_32f_L((const Ipp32f*)pSrc1, src1Stride1,
       src1Stride2, src1Width, src1Height, (const Ipp32f**)ppSrc2,
       src2RoiShift, src2Stride2, src2Length,
       ppDst, dstRoiShift, dstStride2, count);

    printf_va_Ipp32f("2 destination vectors:", dst, 4, 2, status);
    return status;
}
```

The program above produces the following output:

```
2 destination vectors:
22.000000  52.000000  82.000000  44.000000
20.000000  53.000000  86.000000  40.000000
```

When performed on a vector and a transposed matrix (cases 6, 8, 10, 12), the function multiplies all elements in a column of the source matrix by the respective elements in the source vector. Done in loop through all columns in the source matrix, this operation gives the destination vector, which is stored in *pDst*. The following formula applies to all matrix columns *j* and a vector:

$$dst[j] = \sum_{i} src1[i][j] \times src2[i],$$

$0 \le i < src1Height$, $0 \le j < src1Width$.

Note that the number of elements in the source vector *src2Len* must be equal to *src1Height*.

The following example demonstrates how to use the function `ippmMul_tva_32f`. For more information, see also examples in the Getting Started chapter.

Example 5-10 ippmMul_tva_32f

```
IppStatus mul_tva_32f(void) {
    /* Src1 matrix with width=3, height=4, Stride2=2*sizeof(Ipp32f) */
    Ipp32f pSrc1[4*6] = { 1, 0, 2, 0, 3, 0,
                          4, 0, 5, 0, 6, 0,
                          7, 0, 8, 0, 9, 0,
                          2, 0, 4, 0, 6, 0 };
    /* Standard description for Src1 */
    int src1Width   = 3;
    int src1Height  = 4;
    int src1Stride2 = 2*sizeof(Ipp32f);
    int src1Stride1 = 6*sizeof(Ipp32f);

    /* Src2 data: 2 vectors with length=4, Stride2=sizeof(Ipp32f) */
    Ipp32f pSrc2[2*4] = { 1, 6, 3, 2,
                          4, 5, 2, 1 };
    int src2Length  = 4;
    int src2Stride2 = sizeof(Ipp32f);
    int src2Stride0 = 4*sizeof(Ipp32f);
    /*
    // As the first operand is transposed matrix
    // destination vector has length=src1Width=3
    */
    Ipp32f pDst[2*3];
    int dstStride2 = sizeof(Ipp32f);
    int dstStride0 = 3*sizeof(Ipp32f);

    int count  = 2;

    IppStatus status = ippmMul_tva_32f((const Ipp32f*)pSrc1,
        src1Stride1, src1Stride2, src1Width, src1Height,
        (const Ipp32f*)pSrc2, src2Stride0, src2Stride2, src2Length,
        pDst, dstStride0, dstStride2, count);
```

Example 5-10 ippmMul_tva_32f (continued)

```
    printf_va_Ipp32f("2 destination vectors:", pDst, 3, 2, status);
    return status;
}
```

The program above produces the following output:

```
2 destination vectors:
50.000000  64.000000  78.000000
40.000000  53.000000  66.000000
```

When performed on two matrices (cases 13, 17, 21, 25), the function multiplies the elements in a row of the first source matrix by the respective elements in a column of the second source matrix, and sums the products to obtain a new element in the destination matrix. Done in loop through all rows in the first source matrix and all columns in the second source matrix, this operation gives the whole destination matrix, which is stored in *pDst*. For an element in the destination matrix:

$$mdst[i][j] = \sum_{k} src1[i][k] \times src2[k][j],$$

$$0 \le i < src1Height \ , \ 0 \le j < src2Width \ , \ 0 \le k < src1Width \ .$$

Note that the number of columns in the first source matrix *src1Width* must be equal to the number of rows in the second source matrix *src2Height*. The destination matrix has the number of columns equal to *src2Width* and the number of rows equal to *src1Height*.

The following example demonstrates how to use the function `ippmMul_mm_32f`. For more information, see also examples in the Getting Started chapter.

Example 5-11 ippmMul_mm_32f

```
IppStatus mul_mm_32f(void) {
    /* Src1 matrix with width=4 and height=3 */
    Ipp32f pSrc1[3*4] = { 1, 2, 3, 4,
                          5, 6, 7, 8,
                          4, 3, 2, 1 };
    int src1Width  = 4;
    int src1Height = 3;
    int src1Stride2 = sizeof(Ipp32f);
    int src1Stride1 = 4*sizeof(Ipp32f);
```

Example 5-11 ippmMul_mm_32f (continued)

```
    /* Src2 matrix with width=3 and height=4 */
    Ipp32f pSrc2[4*3] = { 1, 5, 4,
                          2, 6, 3,
                          3, 7, 2,
                          4, 8, 1 };
    int src2Width  = 3;
    int src2Height = 4;
    int src2Stride2 = sizeof(Ipp32f);
    int src2Stride1 = 3*sizeof(Ipp32f);

    /*
    // Destination matrix has width=src2Width=3 and height=src1Height=3
    */
    Ipp32f pDst[3*3];
    int dstStride2 = sizeof(Ipp32f);
    int dstStride1 = 3*sizeof(Ipp32f);

    IppStatus status = ippmMul_mm_32f((const Ipp32f*)pSrc1, src1Stride1,
        src1Stride2, src1Width, src1Height, (const Ipp32f*)pSrc2,
        src2Stride1, src2Stride2, src2Width, src2Height,
        pDst, dstStride1, dstStride2);

    printf_m_Ipp32f("Destination matrix:", pDst, 3, 3, status);
    return status;
}
```

The program above produces the following output:

```
Destination matrices:
30.000000  70.000000   20.000000
70.000000  174.000000  60.000000
20.000000  60.000000   30.000000
```

When performed on a transposed matrix and a matrix (cases 14, 18, 22, 26), the function multiplies the elements in a column of the first source matrix by the respective elements in the column of the second source matrix, and sums the products to obtain a new element in the destination matrix. Done in loop through all columns in the first and the second source matrices, this operation gives the whole destination matrix, which is stored in *pDst*. For an element in the destination matrix:

$$dst[i][j] = \sum_{k} src1[k][i] \times src2[k][j],$$

$$0 \le i < src1Width \ , \ 0 \le j < src2Width \ , \ 0 \le k < src1Height \ .$$

Note that the number of rows in the first source matrix *src1Height* must be equal to the number of rows in the second source matrix *src2Height*. The destination matrix has the number of rows equal to *src1Width* and the number of columns equal to *src2Width*.

The following example demonstrates how to use the function ippmMul_tm_32f. To clarify pointer descriptor for transposed matrices, see examples for ippmSub. For more information, see also examples in the Getting Started chapter.

Example 5-12 ippmMul_tm_32f

```
IppStatus mul_tm_32f(void) {
    /* Src1 matrix with width=2 and height=4 */
    Ipp32f pSrc1[4*2] = { 1, 2,
                          3, 4,
                          5, 6,
                          7, 8 };
    int src1Width  = 2;
    int src1Height = 4;
    int src1Stride2 = sizeof(Ipp32f);
    int src1Stride1 = 2*sizeof(Ipp32f);

    /* Src2 matrices have width=3 and height=4 */
    Ipp32f pSrc2[4*3] = { 1, 5, 4,
                          2, 6, 3,
                          3, 7, 2,
                          4, 8, 1 };
    int src2Width  = 3;
    int src2Height = 4;
    int src2Stride2 = sizeof(Ipp32f);
    int src2Stride1 = 3*sizeof(Ipp32f);
```

Example 5-12 ippmMul_tm_32f (continued)

```
    /*
    // As the first operand is transposed matrix
    // destination matrix has width=src2Width=3 and height=src1Width=2
    */
    Ipp32f pDst[2*3];
    int dstStride2 = sizeof(Ipp32f);
    int dstStride1 = 3*sizeof(Ipp32f);

    IppStatus status = ippmMul_tm_32f((const Ipp32f*)pSrc1, src1Stride1,
        src1Stride2, src1Width, src1Height, (const Ipp32f*)pSrc2,
        src2Stride1, src2Stride2, src2Width, src2Height,
        pDst, dstStride1, dstStride2);

    printf_m_Ipp32f("Destination matrix:", pDst, 3, 2, status);
    return status;
}
```

The program above produces the following output:

```
Destination matrix:
50.000000  114.000000  30.000000
60.000000  140.000000  40.00000
```

When performed on a matrix and a transposed matrix (cases 15, 19, 23, 27), the function multiplies the elements in a row of the first source matrix by the respective elements in the a row of the second source matrix, and sums the products to obtain a new element in the destination matrix. Done in loop through all rows in the first source matrix and all rows in the second source matrix, this operation gives the whole destination matrix, which is stored in *pDst*. For an element in the destination matrix:

$$dst[i][j] = \sum_{k} src1[i][k] \times src2[j][k],$$

$$0 \le i < src1Height \ , \ 0 \le j < src2Height \ , \ 0 \le k < src1Width \ .$$

Note that the number of columns in the first source matrix *src1Width* must be equal to the number of columns in the second source matrix *src2Width*. The destination matrix has the number of rows equal to *src1Height* and the number of columns equal to *src2Height*.

When performed on two transposed matrices (cases 16, 20, 24, 28), the function multiplies the elements in a column of the first source matrix by the respective elements in a row of the second source matrix, and sums the products to obtain a new element in the destination matrix. Done in loop through all columns in the first source matrix and all rows in the second source matrix, this operation gives the whole destination matrix, which is stored in *pDst*. For an element in the destination matrix:

$$dst[i][j] = \sum_{k} src1[k][i] \times src2[j][k],$$

$$0 \le i < src1Width \ , \ 0 \le j < src2Height \ , \ 0 \le k < src1Height \ .$$

Note that the number of rows in the first source matrix *src1Height* must be equal to the number of columns in the second source matrix *src2Width*. The destination matrix has the number of rows equal to *src1Width* and the number of columns equal to *src2Height*.

The following example demonstrates how to use the function `ippmMul_tt_32f_P`.

Example 5-13 ippmMul_tt_32f_P

```
IppStatus mul_tt_32f_P(void) {
    /* Src1 source data */
    Ipp32f src1[2*6] = { 9, 0, 0, 8, 0, 7,
                         0, 6, 5, 0, 0, 4 };
    /*
    // Interested nonzero elements are refered by mask using
    // pointer descriptor: Src1 width=3, height=2
    */
    Ipp32f* ppSrc1[2*3] = { src1,   src1+3, src1+5,
                            src1+7, src1+8, src1+11 };
    int src1RoiShift = 0;
    int src1Width  = 3;
    int src1Height = 2;

     /* Src2 source data */
    Ipp32f src2[4*5] = { 0, 7, 0, 2, 0,
                         1, 0, 0, 3, 0,
                         4, 0, 5, 0, 0,
                         6, 0, 0, 0, 8 };
```

Example 5-13 ippmMul_tt_32f_P (continued)

```
    /*
    // Interested nonzero elements are refered by mask using
    // pointer descriptor: Src2 width=2, height=4
    */
    Ipp32f* ppSrc2[4*2] = { src2+1,  src2+3,
                            src2+5,  src2+8,
                            src2+10, src2+12,
                            src2+15, src2+19 };
    int src2RoiShift = 0;
    int src2Width  = 2;
    int src2Height = 4;
    /*
    // As the both operands are transposed matrices
    // destination matrix has width=src2Height=4 and height=src1Width=3
    // Pointer description for destination matrix:
    */
    Ipp32f  dst[3*4];
    Ipp32f* ppDst[3*4] = { dst,   dst+1, dst+2,  dst+3,
                           dst+4, dst+5, dst+6,  dst+7,
                           dst+8, dst+9, dst+10, dst+11 };
    int dstRoiShift = 0;

    IppStatus status = ippmMul_tt_32f_P((const Ipp32f**)ppSrc1,
        src1RoiShift, src1Width, src1Height, (const Ipp32f**)ppSrc2,
        src2RoiShift, src2Width, src2Height, ppDst, dstRoiShift);

    printf_m_Ipp32f_P("Destination matrix:", ppDst, 4, 3, status);
    return status;
}
```

The program above produces the following output:

```
Destination matrix:
75.000000 27.000000 66.000000 102.000000
66.000000 23.000000 57.000000 88.000000
57.000000 19.000000 48.000000 74.000000
```

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.

ippStsSizeErr	Returns an error when the input size parameter is equal to 0.
ippStsStrideMatrixErr	Returns an error when the stride value is not positive or not dividend to size of data type.
ippStsRoiShiftMatrixErr	Returns an error when the roiShift value is negative or not dividend to size of data type.
ippStsCountMatrixErr	Returns an error when the count value is less or equal to zero.
ippStsSizeMatchMatrixErr	Returns an error when the sizes of the source matrices are unsuitable.

Add

Adds matrix to another matrix.

Syntax

Case 1: Matrix - matrix operation

```
IppStatus ippmAdd_mm_32f(const Ipp32f* pSrc1, int src1Stride1, int src1Stride2,
    const Ipp32f* pSrc2, int src2Stride1, int src2Stride2, Ipp32f* pDst,
    int dstStride1, int dstStride2, int width, int height);
IppStatus ippmAdd_mm_64f(const Ipp64f* pSrc1, int src1Stride1, int src1Stride2,
    const Ipp64f* pSrc2, int src2Stride1, int src2Stride2, Ipp64f* pDst,
    int dstStride1, int dstStride2, int width, int height);
IppStatus ippmAdd_mm_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst, int dstRoiShift,
    int width, int height);
IppStatus ippmAdd_mm_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** pDst, int dstRoiShift,
    int width, int height);
```

Case 2: Transposed matrix - matrix operation

```
IppStatus ippmAdd_tm_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp32f* pSrc2, int src2Stride1, int src2Stride2,
    Ipp32f* pDst, int dstStride1, int dstStride2, int width, int height);
IppStatus ippmAdd_tm_64f(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp64f* pSrc2, int src2Stride1, int src2Stride2,
    Ipp64f* pDst, int dstStride1, int dstStride2, int width, int height);
```

```
IppStatus ippmAdd_tm_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst, int dstRoiShift,
    int width, int height);

IppStatus ippmAdd_tm_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst, int dstRoiShift,
    int width, int height);
```

Case 3: Transposed matrix - transposed matrix operation

```
IppStatus ippmAdd_tt_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp32f* pSrc2, int src2Stride1, int src2Stride2,
    Ipp32f* pDst, int dstStride1, int dstStride2, int width, int height);

IppStatus ippmAdd_tt_64f(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp64f* pSrc2, int src2Stride1, int src2Stride2,
    Ipp64f* pDst, int dstStride1, int dstStride2, int width, int height);

IppStatus ippmAdd_tt_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst, int dstRoiShift,
    int width, int height);

IppStatus ippmAdd_tt_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst, int dstRoiShift,
    int width, int height);
```

Case 4: Matrix array - matrix operation

```
IppStatus ippmAdd_mam_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmAdd_mam_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmAdd_mam_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst,
    int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmAdd_mam_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst,
    int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmAdd_mam_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int width, int height, int count);
```

```
IppStatus ippmAdd_mam_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int width, int height, int count);
```

Case 5: Transposed matrix array - matrix operation

```
IppStatus ippmAdd_tam_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmAdd_tam_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmAdd_tam_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst,
    int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmAdd_tam_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst,
    int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmAdd_tam_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmAdd_tam_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int width, int height, int count);
```

Case 6: Matrix array - transposed matrix operation

```
IppStatus ippmAdd_mat_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmAdd_mat_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmAdd_mat_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst,
    int dstRoiShift, int dstStride0, int width, int height, int count);
```

```
IppStatus ippmAdd_mat_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst,
    int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmAdd_mat_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmAdd_mat_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int width, int height, int count);
```

Case 7: Transposed matrix array - transposed matrix operation

```
IppStatus ippmAdd_tat_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmAdd_tat_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmAdd_tat_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst,
    int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmAdd_tat_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst,
    int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmAdd_tat_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmAdd_tat_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int width, int height, int count);
```

Case 8: Matrix array - matrix array operation

```
IppStatus ippmAdd_mama_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride0,
    int src2Stride1, int src2Stride2, Ipp32f* pDst, int dstStride0,
    int dstStride1, int dstStride2, int width, int height, int count);
```

```
IppStatus ippmAdd_mama_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride0,
    int src2Stride1, int src2Stride2, Ipp64f* pDst, int dstStride0,
    int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmAdd_mama_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp32f** ppDst, int dstRoiShift, int dstStride0, int width, int height,
    int count);

IppStatus ippmAdd_mama_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp64f** ppDst, int dstRoiShift, int dstStride0, int width, int height,
    int count);

IppStatus ippmAdd_mama_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Stride1, int src2Stride2, Ipp32f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmAdd_mama_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Stride1, int src2Stride2, Ipp64f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int width, int height, int count);
```

Case 9: Transposed matrix array - matrix array operation

```
IppStatus ippmAdd_tama_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride0,
    int src2Stride1, int src2Stride2, Ipp32f* pDst, int dstStride0,
    int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmAdd_tama_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride0,
    int src2Stride1, int src2Stride2, Ipp64f* pDst, int dstStride0,
    int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmAdd_tama_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp32f** ppDst, int dstRoiShift, int dstStride0, int width, int height,
    int count);

IppStatus ippmAdd_tama_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp64f** ppDst, int dstRoiShift, int dstStride0, int width, int height,
    int count);
```

```
IppStatus ippmAdd_tama_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Stride1, int src2Stride2, Ipp32f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmAdd_tama_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Stride1, int src2Stride2, Ipp64f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int width, int height, int count);
```

Case 10: Transposed matrix array - transposed matrix array operation

```
IppStatus ippmAdd_tata_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride0,
    int src2Stride1, int src2Stride2, Ipp32f* pDst, int dstStride0,
    int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmAdd_tata_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride0,
    int src2Stride1, int src2Stride2, Ipp64f* pDst, int dstStride0,
    int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmAdd_tata_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp32f** ppDst, int dstRoiShift, int dstStride0, int width, int height,
    int count);

IppStatus ippmAdd_tata_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp64f** ppDst, int dstRoiShift, int dstStride0, int width, int height,
    int count);

IppStatus ippmAdd_tata_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Stride1, int src2Stride2, Ipp32f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmAdd_tata_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Stride1, int src2Stride2, Ipp64f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int width, int height, int count);
```

Parameters

pSrc1, ppSrc1	Pointer to the first source matrix or array of matrices.
src1Stride0	Stride between the matrices in the first source array.
src1Stride1	Stride between the rows in the first source matrix(ces).

src1Stride2	Stride between the elements in the first source matrix(ces).
src1RoiShift	ROI shift in the first source matrix(ces).
pSrc2, ppSrc2	Pointer to the second source matrix or array of matrices.
src2Stride0	Stride between the matrices in the second source array.
src2Stride1	Stride between the rows in the second source matrix(ces).
src2Stride2	Stride between the elements in the second source matrix(ces).
src2RoiShift	ROI shift in the second source matrix(ces).
pDst, ppDst	Pointer to the destination matrix or array of matrices.
dstStride0	Stride between the matrices in the destination array.
dstStride1	Stride between the rows in the destination matrix.
dstStride2	Stride between the elements in the destination matrix.
dstRoiShift	ROI shift in the destination matrix.
width	Matrix width.
height	Matrix height.
count	Number of matrices in the array.

Description

The function `ippmAdd` is declared in the `ippm.h` header file.

When performed on two matrices (cases 1, 4, 8), the function adds together the respective elements of the first and the second source matrices and stores the result in *pDst*:

$dst[i][j] = src1[i][j] + src2[i][j]$,

$0 \le i < height$, $0 \le j < width$.

When performed on a transposed matrix and a matrix (cases 2, 5, 9), the function adds together the respective elements of the transposed matrix and the second source matrix and stores the result in *pDst*:

$dst[i][j] = src1[j][i] + src2[i][j]$,

$0 \le i < height$, $0 \le j < width$.

Note that the first source matrix must have the number of rows equal to *width* and the number of columns equal to *height*.

The following example demonstrates how to use the function `ippmAdd_tm_32f`. To clarify pointer descriptor for transposed matrices, see examples for `ippmSub`. For more information, see also examples in the Getting Started chapter.

Example 5-14 ippmAdd_tm_32f

```
IppStatus add_tm_32f(void) {
    /* Src1 matrix with width=4 and height=3 */
    Ipp32f pSrc1[3*4] = { 1, 2, 3, 4,
                          5, 6, 7, 8,
                          4, 3, 2, 1 };
    int src1Stride2 = sizeof(Ipp32f);
    int src1Stride1 = 4*sizeof(Ipp32f);

    /* Src2 and Dst matrices have width=3 and height=4 */
    Ipp32f pSrc2[4*3] = { 1, 5, 4,
                          2, 6, 3,
                          3, 7, 2,
                          4, 8, 1 };
    int src2Stride2 = sizeof(Ipp32f);
    int src2Stride1 = 3*sizeof(Ipp32f);
    Ipp32f pDst[4*3]; /* Destination location */
    int dstStride2 = sizeof(Ipp32f);
    int dstStride1 = 3*sizeof(Ipp32f);

    int width  = 3;
    int height = 4;

    IppStatus status = ippmAdd_tm_32f((const Ipp32f*)pSrc1, src1Stride1,
         src1Stride2, (const Ipp32f*)pSrc2, src2Stride1, src2Stride2,
         pDst, dstStride1, dstStride2, width, height);

    printf_m_Ipp32f("Destination matrix:", pDst, 3, 4, status);
    return status;
}
```

The program above produces the following output:

```
Destination matrix:
2.000000  10.000000  8.000000
4.000000  12.000000  6.000000
6.000000  14.000000  4.000000
8.000000  16.000000  2.000000
```

When performed on a matrix array and a transposed matrix (case 6), the function adds together the respective elements of the first matrix in the array and the transposed matrix and stores the result in *pDst*:

$dst[i][j] = src1[i][j] + src2[j][i]$,

$0 \le i < height$, $0 \le j < width$.

Note that the second source matrix must have the number of rows equal to *width* and the number of columns equal to *height*.

When performed on two transposed matrices (cases 3, 7, 10), the function adds together the respective elements of the first and the second transposed matrices and stores the result in *pDst*:

$dst[i][j] = src1[j][i] + src2[j][i]$,

$0 \le i < height$, $0 \le j < width$.

Note that both source matrices must have the number of rows equal to *width* and the number of columns equal to *height*.

The following example demonstrates how to use the function `ippmAdd_tt_32f`. To clarify pointer descriptor for transposed matrices, see examples for `ippmSub`. For more information, see also examples in the Getting Started chapter

.

Example 5-15 ippmAdd_tt_32f

```
IppStatus add_tt_32f(void) {
    /* Src1 and Src2 matrices have width=4 and height=3 */
    Ipp32f pSrc1[3*4] = { 1, 2, 3, 1,
                          4, 5, 6, 1,
                          7, 8, 9, 1 };
    int src1Stride2 = sizeof(Ipp32f);
    int src1Stride1 = 4*sizeof(Ipp32f);

    Ipp32f pSrc2[3*4] = { 9, 8, 7, -1,
                          6, 5, 4, -1,
                          3, 2, 1, -1 };
    int src2Stride2 = sizeof(Ipp32f);
    int src2Stride1 = 4*sizeof(Ipp32f);

    /* Dst matrices have width=3 and height=4 */
    Ipp32f pDst[4*3];

    int dstStride2 = sizeof(Ipp32f);
    int dstStride1 = 3*sizeof(Ipp32f);


    int width  = 3;
    int height = 4;
    IppStatus status = ippmAdd_tt_32f((const Ipp32f*)pSrc1, src1Stride1,
         src1Stride2, (const Ipp32f*)pSrc2, src2Stride1, src2Stride2,
         pDst, dstStride1, dstStride2, width, height);

    printf_m_Ipp32f("Destination matrix:", pDst, 3, 4, status);
    return status;
}
```

The program above produces the following output:

```
Destination matrix:
10.000000  10.000000  10.000000
10.000000  10.000000  10.000000
10.000000  10.000000  10.000000
0.000000   0.000000   0.000000
```

Return Values

ippStsOk	Returns no error.
ippStsNullPtrErr	Returns an error when at least one input pointer is NULL.
ippStsSizeErr	Returns an error when the input size parameter is equal to 0.
ippStsStrideMatrixErr	Returns an error when the stride value is not positive or not dividend to size of data type.
ippStsRoiShiftMatrixErr	Returns an error when the roiShift value is negative or not dividend to size of data type.
ippStsCountMatrixErr	Returns an error when the count value is less or equal to zero.

Sub

Subtracts matrix from another matrix.

Syntax

Case 1: Matrix - matrix operation

```
IppStatus ippmSub_mm_32f(const Ipp32f* pSrc1, int src1Stride1, int src1Stride2,
    const Ipp32f* pSrc2, int src2Stride1, int src2Stride2, Ipp32f* pDst,
    int dstStride1, int dstStride2, int width, int height);
IppStatus ippmSub_mm_64f(const Ipp64f* pSrc1, int src1Stride1, int src1Stride2,
    const Ipp64f* pSrc2, int src2Stride1, int src2Stride2, Ipp64f* pDst,
    int dstStride1, int dstStride2, int width, int height);
IppStatus ippmSub_mm_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst, int dstRoiShift,
    int width, int height);
IppStatus ippmSub_mm_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst, int dstRoiShift,
    int width, int height);
```

Case 2: Transposed matrix - matrix operation

```
IppStatus ippmSub_tm_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp32f* pSrc2, int src2Stride1, int src2Stride2,
    Ipp32f* pDst, int dstStride1, int dstStride2, int width, int height);
```

```
IppStatus ippmSub_tm_64f(const Ipp64f* pSrc1, int src1Stride1,
   int src1Stride2, const Ipp64f* pSrc2, int src2Stride1, int src2Stride2,
   Ipp64f* pDst, int dstStride1, int dstStride2, int width, int height);

IppStatus ippmSub_tm_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
   const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst, int dstRoiShift,
   int width, int height);

IppStatus ippmSub_tm_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
   const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst, int dstRoiShift,
   int width, int height);
```

Case 3: Matrix - transposed matrix operation

```
IppStatus ippmSub_mt_32f(const Ipp32f* pSrc1, int src1Stride1,
   int src1Stride2, const Ipp32f* pSrc2, int src2Stride1, int src2Stride2,
   Ipp32f* pDst, int dstStride1, int dstStride2, int width, int height);

IppStatus ippmSub_mt_64f(const Ipp64f* pSrc1, int src1Stride1,
   int src1Stride2, const Ipp64f* pSrc2, int src2Stride1, int src2Stride2,
   Ipp64f* pDst, int dstStride1, int dstStride2, int width, int height);

IppStatus ippmSub_mt_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
   const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst, int dstRoiShift,
   int width, int height);

IppStatus ippmSub_mt_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
   const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst, int dstRoiShift,
   int width, int height);
```

Case 4: Transposed matrix - transposed matrix operation

```
IppStatus ippmSub_tt_32f(const Ipp32f* pSrc1, int src1Stride1,
   int src1Stride2, const Ipp32f* pSrc2, int src2Stride1, int src2Stride2,
   Ipp32f* pDst, int dstStride1, int dstStride2, int width, int height);

IppStatus ippmSub_tt_64f(const Ipp64f* pSrc1, int src1Stride1,
   int src1Stride2, const Ipp64f* pSrc2, int src2Stride1, int src2Stride2,
   Ipp64f* pDst, int dstStride1, int dstStride2, int width, int height);

IppStatus ippmSub_tt_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
   const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst, int dstRoiShift,
   int width, int height);

IppStatus ippmSub_tt_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
   const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst, int dstRoiShift,
   int width, int height);
```

Case 5: Matrix - matrix array operation

```
IppStatus ippmSub_mma_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp32f* pSrc2, int src2Stride0, int src2Stride1,
    int src2Stride2, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmSub_mma_64f(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp64f* pSrc2, int src2Stride0, int src2Stride1,
    int src2Stride2, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmSub_mma_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp32f** ppDst,
    int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmSub_mma_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp64f** ppDst,
    int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmSub_mma_32f_L(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride1,
    int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmSub_mma_64f_L(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride1,
    int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int width, int height, int count);
```

Case 6: Transposed matrix - matrix array operation

```
IppStatus ippmSub_tma_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp32f* pSrc2, int src2Stride0, int src2Stride1,
    int src2Stride2, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmSub_tma_64f(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp64f* pSrc2, int src2Stride0, int src2Stride1,
    int src2Stride2, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmSub_tma_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp32f** ppDst,
    int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmSub_tma_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp64f** ppDst,
    int dstRoiShift, int dstStride0, int width, int height, int count);
```

```
IppStatus ippmSub_tma_32f_L(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride1,
    int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmSub_tma_64f_L(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride1,
    int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int width, int height, int count);
```

Case 7: Matrix - transposed matrix array operation

```
IppStatus ippmSub_mta_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp32f* pSrc2, int src2Stride0, int src2Stride1,
    int src2Stride2, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmSub_mta_64f(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp64f* pSrc2, int src2Stride0, int src2Stride1,
    int src2Stride2, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmSub_mta_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp32f** ppDst,
    int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmSub_mta_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp64f** ppDst,
    int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmSub_mta_32f_L(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride1,
    int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmSub_mta_64f_L(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride1,
    int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int width, int height, int count);
```

Case 8: Transposed matrix - transposed matrix array operation

```
IppStatus ippmSub_tta_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp32f* pSrc2, int src2Stride0, int src2Stride1,
    int src2Stride2, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);
```

```
IppStatus ippmSub_tta_64f(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp64f* pSrc2, int src2Stride0, int src2Stride1,
    int src2Stride2, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmSub_tta_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp32f** ppDst,
    int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmSub_tta_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0, Ipp64f** ppDst,
    int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmSub_tta_32f_L(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride1,
    int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmSub_tta_64f_L(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride1,
    int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int width, int height, int count);
```

Case 9: Matrix array - matrix operation

```
IppStatus ippmSub_mam_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmSub_mam_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmSub_mam_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst,
    int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmSub_mam_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst,
    int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmSub_mam_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int width, int height, int count);
```

```
IppStatus ippmSub_mam_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
   int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride1,
   int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
   int dstStride2, int width, int height, int count);
```

Case 10: Transposed matrix array - matrix operation

```
IIppStatus ippmSub_tam_32f(const Ipp32f* pSrc1, int src1Stride0,
   int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride1,
   int src2Stride2, Ipp32f* pDst, int dstStride0, int dstStride1,
   int dstStride2, int width, int height, int count);

IppStatus ippmSub_tam_64f(const Ipp64f* pSrc1, int src1Stride0,
   int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride1,
   int src2Stride2, Ipp64f* pDst, int dstStride0, int dstStride1,
   int dstStride2, int width, int height, int count);

IppStatus ippmSub_tam_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
   int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst,
   int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmSub_tam_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
   int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst,
   int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmSub_tam_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
   int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride1,
   int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
   int dstStride2, int width, int height, int count);

IppStatus ippmSub_tam_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
   int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride1,
   int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
   int dstStride2, int width, int height, int count);
```

Case 11: Matrix array - transposed matrix operation

```
IppStatus ippmSub_mat_32f(const Ipp32f* pSrc1, int src1Stride0,
   int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride1,
   int src2Stride2, Ipp32f* pDst, int dstStride0, int dstStride1,
   int dstStride2, int width, int height, int count);

IppStatus ippmSub_mat_64f(const Ipp64f* pSrc1, int src1Stride0,
   int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride1,
   int src2Stride2, Ipp64f* pDst, int dstStride0, int dstStride1,
   int dstStride2, int width, int height, int count);
```

```
IppStatus ippmSub_mat_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst,
    int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmSub_mat_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst,
    int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmSub_mat_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmSub_mat_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int width, int height, int count);
```

Case 12: Transposed matrix array - transposed matrix operation

```
IppStatus ippmSub_tat_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp32f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmSub_tat_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp64f* pDst, int dstStride0, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmSub_tat_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst,
    int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmSub_tat_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst,
    int dstRoiShift, int dstStride0, int width, int height, int count);

IppStatus ippmSub_tat_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int width, int height, int count);

IppStatus ippmSub_tat_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride1,
    int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride1,
    int dstStride2, int width, int height, int count);
```

Case 13: Matrix array - matrix array operation

```
IppStatus ippmSub_mama_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride0,
    int src2Stride1, int src2Stride2, Ipp32f* pDst, int dstStride0,
    int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmSub_mama_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride0,
    int src2Stride1, int src2Stride2, Ipp64f* pDst, int dstStride0,
    int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmSub_mama_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp32f** ppDst, int dstRoiShift, int dstStride0, int width, int height,
    int count);

IppStatus ippmSub_mama_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp64f** ppDst, int dstRoiShift, int dstStride0, int width, int height,
    int count);

IppStatus ippmSub_mama_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Stride1, int src2Stride2, Ipp32f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmSub_mama_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Stride1, int src2Stride2, Ipp64f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int width, int height, int count);
```

Case 14: Transposed matrix array - matrix array operation

```
IppStatus ippmSub_tama_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride0,
    int src2Stride1, int src2Stride2, Ipp32f* pDst, int dstStride0,
    int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmSub_tama_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride0,
    int src2Stride1, int src2Stride2, Ipp64f* pDst, int dstStride0,
    int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmSub_tama_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp32f** ppDst, int dstRoiShift, int dstStride0, int width, int height,
    int count);
```

```
IppStatus ippmSub_tama_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp64f** ppDst, int dstRoiShift, int dstStride0, int width, int height,
    int count);

IppStatus ippmSub_tama_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Stride1, int src2Stride2, Ipp32f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmSub_tama_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Stride1, int src2Stride2, Ipp64f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int width, int height, int count);
```

Case 15: Matrix array - transposed matrix array operation

```
IppStatus ippmSub_mata_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride0,
    int src2Stride1, int src2Stride2, Ipp32f* pDst, int dstStride0,
    int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmSub_mata_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride0,
    int src2Stride1, int src2Stride2, Ipp64f* pDst, int dstStride0,
    int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmSub_mata_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp32f** ppDst, int dstRoiShift, int dstStride0, int width, int height,
    int count);

IppStatus ippmSub_mata_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp64f** ppDst, int dstRoiShift, int dstStride0, int width, int height,
    int count);

IppStatus ippmSub_mata_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Stride1, int src2Stride2, Ipp32f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int width, int height, int count);

IppStatus ippmSub_mata_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Stride1, int src2Stride2, Ipp64f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int width, int height, int count);
```

Case 16: Transposed matrix array - transposed matrix array operation

```
IppStatus ippmSub_tata_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride0,
    int src2Stride1, int src2Stride2, Ipp32f* pDst, int dstStride0,
    int dstStride1, int dstStride2, int width, int height, int count);
IppStatus ippmSub_tata_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride0,
    int src2Stride1, int src2Stride2, Ipp64f* pDst, int dstStride0,
    int dstStride1, int dstStride2, int width, int height, int count);
IppStatus ippmSub_tata_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp32f** ppDst, int dstRoiShift, int dstStride0, int width, int height,
    int count);
IppStatus ippmSub_tata_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp64f** ppDst, int dstRoiShift, int dstStride0, int width, int height,
    int count);
IppStatus ippmSub_tata_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Stride1, int src2Stride2, Ipp32f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int width, int height, int count);
IppStatus ippmSub_tata_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Stride1, int src2Stride2, Ipp64f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int width, int height, int count);
```

Parameters

pSrc1, *ppSrc1*	Pointer to the first source matrix or array of matrices.
src1Stride0	Stride between the matrices in the first source array.
src1Stride1	Stride between the rows in the first source matrix(ces).
src1Stride2	Stride between the elements in the first source matrix(ces).
src1RoiShift	ROI shift in the first source matrix(ces).
pSrc2, *ppSrc2*	Pointer to the second source matrix or array of matrices.
src2Stride0	Stride between the matrices in the second source array.
src2Stride1	Stride between the rows in the second source matrix(ces).
src2Stride2	Stride between the elements in the second source matrix(ces).

src2RoiShift	ROI shift in the second source matrix(ces).
pDst, ppDst	Pointer to the destination matrix or array of matrices.
dstStride0	Stride between the matrices in the destination array.
dstStride1	Stride between the rows in the destination matrix.
dstStride2	Stride between the elements in the destination matrix.
dstRoiShift	ROI shift in the destination matrix.
width	Matrix width.
height	Matrix height.
count	Number of matrices in the array.

Description

The function ippmSub is declared in the ippm.h header file.

When performed on two matrices (cases 1, 5, 9, 13), the function subtracts an element of the second source matrix from the respective element of the first source matrix and stores the result in *pDst*. This operation is done in loop through all matrix elements:

$dst[i][j] = src1[i][j] - src2[i][j],$

$0 \le i < height,\ 0 \le j < width$.

When performed on a transposed matrix and a matrix (cases 2, 6, 10, 14), the function subtracts an element of the second source matrix from the respective element of the first source matrix and stores the result in *pDst*. This operation is done in loop through all matrix elements:

$dst[i][j] = src1[j][i] - src2[i][j],$

$0 \le i < height, 0 \le j < width.$

Note that the transposed matrix must have the number of rows equal to *width* and the number of columns equal to *height*.

The following example demonstrates how to use the function ippmSub_tm_32f_P. To clarify other cases, see examples for the ippmAdd function. For more information, see also examples in the Getting Started chapter.

Example 5-16 ippmSub_tm_32f_P

```
IppStatus sub_tm_32f_P(void) {
    /* Src1 source data */
    Ipp32f src1[2*6] = { 9, 0, 0, 8, 0, 7,
                         0, 6, 5, 0, 0, 4 };
    /*
    // Interested nonzero elements are refered by mask using
    // pointer descriptor: Src1 width=3, height=2
    */
    Ipp32f* ppSrc1[2*3] = { src1,   src1+3, src1+5,
                            src1+7, src1+8, src1+11 };
    int src1RoiShift = 0;

     /* Src2 source data */
    Ipp32f src2[3*6] = { 7, 0, 0, 0, 0, 1,
                         0, 2, 1, 0, 0, 0,
                         0, 0, 4, 0, 3, 0};

    /*
    // Interested nonzero elements are refered by mask using
    // Pointer descriptor: Src2 width=2, height=3
    */
    Ipp32f* ppSrc2[3*2] = { src2,    src2+5,
                            src2+7,  src2+8,
                            src2+14, src2+16 };
    int src2RoiShift = 0;
    /*
    // Pointer description for destination matrix:
    // Dst width=2, height=3
    */
    Ipp32f  dst[3*2];
    Ipp32f* ppDst[3*2] = { dst,   dst+1,
                           dst+2, dst+3,
                           dst+4, dst+5 };

    int dstRoiShift = 0;
    int width  = 2;
    int height = 3;
```

Example 5-16 ippmSub_tm_32f_P (continued)

```
    IppStatus status = ippmSub_tm_32f_P((const Ipp32f**)ppSrc1,
        src1RoiShift, (const Ipp32f**)ppSrc2, src2RoiShift,
        ppDst, dstRoiShift, width, height );

    printf_m_Ipp32f_P("Destination matrix:", ppDst, 2, 3, status);
    return status;
}
```

The program above produces the following output:

```
Destination matrix:
2.000000  10.000000  8.000000
4.000000  12.000000  6.000000
6.000000  14.000000  4.000000
8.000000  16.000000  2.000000
```

When performed on a matrix array and a transposed matrix (cases 3, 7, 11, 15), the function subtracts an element of the transposed source matrix from the respective element of the first source matrix and stores the result in *pDst*. This operation is done in loop through all matrix elements:

$dst[i][j] = src1[i][j] - src2[j][i]$,

$0 \le i < height$, $0 \le j < width$.

Note that the transposed matrix must have the number of rows equal to *width* and the number of columns equal to *height*.

When performed on two transposed matrices (cases 4, 8, 12, 16), the function subtracts an element of the second transposed matrix from the respective element of the first transposed matrix and stores the result in *pDst*. This operation is done in loop through all matrix elements:

$dst[i][j] = src1[j][i] - src2[j][i]$,

$0 \le i < height$, $0 \le j < width$.

Note that both transposed matrices must have the number of rows equal to *width* and the number of columns equal to *height*.

The following example demonstrates how to use the function ippmSub_tt_32f_P.

Example 5-17 ippmSub_tt_32f_P

```
IppStatus sub_tt_32f_P(void) {
    /* Src1 source data */
    Ipp32f src1[2*6] = { 9, 0, 0, 8, 0, 7,
                         0, 6, 5, 0, 0, 4 };
    /*
    // Interested nonzero elements are refered by mask using
    // pointer descriptor: Src1 width=3, height=2
    */
    Ipp32f* ppSrc1[2*3] = { src1,   src1+3, src1+5,
                            src1+7, src1+8, src1+11 };
    int src1RoiShift = 0;

     /* Src2 source data */
    Ipp32f src2[2*6] = { 0, 7, 0, 2, 4, 0,
                         1, 1, 0, 3, 0, 0 };

    /*
    // Interested nonzero elements are refered by mask using
    // pointer descriptor: Src2 width=3, height=2
    */
    Ipp32f* ppSrc2[2*3] = { src2+1, src2+3, src2+4,
                            src2+6, src2+7, src2+9 };
    int src2RoiShift = 0;
    /*
    // Pointer description for destination matrix:
    // Dst width=2, height=3
    */
    Ipp32f  dst[3*2];
    Ipp32f* ppDst[3*2] = { dst,   dst+1,
                           dst+2, dst+3,
                           dst+4, dst+5 };
    int dstRoiShift = 0;
    int width  = 2;
    int height = 3;
```

Example 5-17 ippmSub_tt_32f_P (continued)

```
    IppStatus status = ippmSub_tt_32f_P((const Ipp32f**)ppSrc1,
        src1RoiShift, (const Ipp32f**)ppSrc2, src2RoiShift,
        ppDst, dstRoiShift, width, height );

    printf_m_Ipp32f_P("Destination matrix:", ppDst, 2, 3, status);
    return status;
}
```

The program above produces the following output:

```
Destination matrix:
2.000000  10.000000  8.000000
4.000000  12.000000  6.000000
6.000000  14.000000  4.000000
8.000000  16.000000  2.000000
```

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the input size parameter is equal to 0.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not dividend to size of data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the roiShift value is negative or not dividend to size of data type.
`ippStsCountMatrixErr`	Returns an error when the count value is less or equal to zero.

Gaxpy

Performs the "gaxpy" operation on a matrix.

Syntax

Case 1: Matrix - vector operation

```
IppStatus ippmGaxpy_mv_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp32f* pSrc2,
    int src2Stride2, int src2Len, const Ipp32f* pSrc3, int src2Stride3,
    int src3Len, Ipp32f* pDst, int dstStride2);

IppStatus ippmGaxpy_mv_64f(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp64f* pSrc2,
    int src2Stride2, int src2Len, const Ipp64f* pSrc3, int src3Stride2,
    int src3Len, Ipp64f* pDst, int dstStride2);

IppStatus ippmGaxpy_mv_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Len, const Ipp32f** ppSrc3, int src3RoiShift, int src3Len,
    Ipp32f** ppDst, int dstRoiShift);

IppStatus ippmGaxpy_mv_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Len, const Ipp64f** ppSrc3, int src3RoiShift, int src3Len,
    Ipp64f** ppDst, int dstRoiShift);
```

Case 2: Matrix - vector array operation

```
IppStatus ippmGaxpy_mva_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp32f* pSrc2,
    int src2Stride0, int src2Stride2, int src2Len, const Ipp32f* pSrc3,
    int src3Stride0, int src3Stride2, int src3Len, Ipp32f* pDst,
    int dstStride0, int dstStride2, int count);

IppStatus ippmGaxpy_mva_64f(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp64f* pSrc2,
    int src2Stride0, int src2Stride2, int src2Len, const Ipp64f* pSrc3,
    int src3Stride0, int src3Stride2, int src3Len, Ipp64f* pDst,
    int dstStride0, int dstStride2, int count);

IppStatus ippmGaxpy_mva_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Stride0, int src2Len, const Ipp32f** ppSrc3, int src3RoiShift,
    int src3Stride0, int src3Len, Ipp32f** ppDst, int dstRoiShift,
    int dstStride0, int count);
```

```
IppStatus ippmGaxpy_mva_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Width, int src1Height, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Stride0, int src2Len, const Ipp64f** ppSrc3, int src3RoiShift,
    int src3Stride0, int src3Len, Ipp64f** ppDst, int dstRoiShift,
    int dstStride0, int count);
IppStatus ippmGaxpy_mva_32f_L(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp32f** ppSrc2,
    int src2RoiShift, int src2Stride2, int src2Len, const Ipp32f** ppSrc3,
    int src3RoiShift, int src3Stride2, int src3Len, Ipp32f** ppDst,
    int dstRoiShift, int dstStride2, int count);
IppStatus ippmGaxpy_mva_64f_L(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, int src1Width, int src1Height, const Ipp64f** ppSrc2,
    int src2RoiShift, int src2Stride2, int src2Len, const Ipp64f** ppSrc3,
    int src3RoiShift, int src3Stride2, int sr32Len, Ipp64f** ppDst,
    int dstRoiShift, int dstStride2, int count;)
```

Parameters

pSrc1, ppSrc1	Pointer to the source matrix or array of matrices.
src1Stride0	Stride between the matrices in the source matrix array.
src1Stride1	Stride between the rows in the source matrix(ces).
src1Stride2	Stride between the elements in the source matrix(ces).
src1RoiShift	ROI shift in the source matrix(ces).
src1Width	Matrix width.
src1Height	Matrix height.
pSrc2, ppSrc2	Pointer to the first source vector or array of vectors.
src2Stride0	Stride between the vectors in the first source vector array.
src2Stride2	Stride between the elements in the first source vector(s).
src2RoiShift	ROI shift in the first source vector(s).
src2Len	Length of the first source vector(s).
pSrc3, ppSrc3	Pointer to the second source vector or array of vectors.
src3Stride0	Stride between the vectors in the second source vector array.
src3Stride2	Stride between the elements in the second source vector(s).
src3RoiShift	ROI shift in the second source vector(s).

src3Len	Length of the second source vector(s).
pDst, ppDst	Pointer to the destination vector or array of vectors.
dstStride0	Stride between the vectors in the destination array.
dstStride2	Stride between the elements in the destination vector(s).
dstRoiShift	ROI shift in the destination vector(s).
count	Number of objects in the array.

Description

The function `ippmGaxpy` is declared in the `ippm.h` header file. The function performs the "gaxpy" operation on a matrix by:

1. composing dot product of a row in the first source matrix and the second source vector
2. adding the result to the respective element in the third source vector, and
3. storing the result in the respective element in the destination:

$$t[i] = \sum_{i} (src1[i][j] \cdot src2[j]) + src3[i],$$

$$0 \le i < src1Height, 0 \le j < src1Width.$$

The function iterates consequently through all the rows of the matrices in question.

Note that the number of elements in the second source vector *src2Len* must be equal to *src1Width* and the number of elements in the third source vector *src3Len* must be equal to *src1Height*.

The following example demonstrates how to use the function `ippmGaxpy_mva_32f`. For more information, see also examples in the Getting Started chapter.

Example 5-18 ippmGaxpy_mva_32f

```
IppStatus gaxpy_mva_32f(void) {
    /* Src1 matrix with width=3, height=4, Stride2=2*sizeof(Ipp32f) */
    Ipp32f pSrc1[4*6] = { 1, 0, 2, 0, 3, 0,
                          4, 0, 5, 0, 6, 0,
                          7, 0, 8, 0, 9, 0,
                          2, 0, 4, 0, 6, 0 };
    /* Src2 data: 2 vectors with length=3, Stride2=sizeof(Ipp32f) */
    Ipp32f pSrc2[2*3] = { 1, 6, 3,
                          4, 5, 2 };

    /* Src3 data: 2 vectors with length=4, Stride2=sizeof(Ipp32f) */
    Ipp32f pSrc3[2*4] = { -11, -16, -13, -17,
                           -4, -15, -12, -18 };

    /* Standard description for Src1 */
    int src1Width   = 3;
    int src1Height  = 4;
    int src1Stride2 = 2 * sizeof(Ipp32f);
    int src1Stride1 = 6 * sizeof(Ipp32f);

    /* Standard description for Src2 */
    int src2Length = 3;
    int src2Stride2  = sizeof(Ipp32f);
    int src2Stride0  = 3*sizeof(Ipp32f);

    /* Standard description for Src3 */
    int src3Length = 4;
    int src3Stride2  = sizeof(Ipp32f);
    int src3Stride0  = 4*sizeof(Ipp32f);

    /*
    // Destination vector has length=src1Height=src3Length=4
    // Standard description for Dst:
    */
    Ipp32f pDst[2*4];
    int dstStride2  = sizeof(Ipp32f);
    int dstStride0  = 4*sizeof(Ipp32f);

    int count  = 2;
```

Example 5-18 ippmGaxpy_mva_32f (continued)

```
    IppStatus status = ippmGaxpy_mva_32f((const Ipp32f*)pSrc1,
        src1Stride1, src1Stride2, src1Width, src1Height,
        (const Ipp32f*)pSrc2, src2Stride0, src2Stride2, src2Length,
        (const Ipp32f*)pSrc3, src3Stride0, src3Stride2, src3Length,
        pDst, dstStride0, dstStride2, count);

    printf_va_Ipp32f("2 destination vectors:", pDst, 4, 2, status);
    return status;
}
```

The program above produces the following output:

```
2 destination vectors:
11.000000  36.000000  69.000000  27.000000
16.000000  38.000000  74.000000  22.000000
```

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the input size parameter is equal to 0.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not dividend to size of data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the roiShift value is negative or not dividend to size of data type.
`ippStsCountMatrixErr`	Returns an error when the count value is less or equal to zero.
`ippStsSizeMatchMatrixErr`	Returns an error when the sizes of the source matrices are unsuitable.

Linear System Solution Functions

6

This chapter describes Intel® Integrated Performance Primitives (Intel® IPP) functions for small matrices that decompose square matrices and solve system of linear equations by back substitution.

Table 6-1 Linear System Solution functions

Function Base Name	Operation
LUDecomp	Decomposes square matrix into product of upper and lower triangular matrices.
LUBackSubst	Solves system of linear equations with LU-factored square matrix.
CholeskyDecomp	Performs Cholesky decomposition of a symmetric positive definite square matrix.
CholeskyBackSubst	Solves system of linear equations using the Cholesky triangular factor.

LUDecomp

Decomposes square matrix into product of upper and lower triangular matrices.

Syntax

Case 1: Matrix operation

```
IppStatus ippmLUDecomp_m_32f(const Ipp32f* pSrc, int srcStride1,
    int srcStride2, int* pDstIndex, Ipp32f* pDst, int dstStride1,
    int dstStride2, int widthHeight);

IppStatus ippmLUDecomp_m_64f(const Ipp64f* pSrc, int srcStride1,
    int srcStride2, int* pDstIndex, Ipp64f* pDst, int dstStride1,
    int dstStride2, int widthHeight);

IppStatus ippmLUDecomp_m_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    int* pDstIndex, Ipp32f** ppDst, int dstRoiShift, int widthHeight);

IppStatus ippmLUDecomp_m_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    int* pDstIndex, Ipp64f** ppDst, int dstRoiShift, int widthHeight);
```

Case 2: Matrix array operation

```
IppStatus ippmLUDecomp_ma_32f(const Ipp32f* pSrc, int srcStride0,
    int srcStride1, int srcStride2, int* pDstIndex, Ipp32f* pDst,
    int dstStride0, int dstStride1, int dstStride2, int widthHeight, int count);

IppStatus ippmLUDecomp_ma_64f(const Ipp64f* pSrc, int srcStride0,
    int srcStride1, int srcStride2, int* pDstIndex, Ipp64f* pDst,
    int dstStride0, int dstStride1, int dstStride2, int widthHeight, int count);

IppStatus ippmLUDecomp_ma_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride0, int* pDstIndex, Ipp32f** ppDst, int dstRoiShift,
    int dstStride0, int widthHeight, int count);

IppStatus ippmLUDecomp_ma_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride0, int* pDstIndex, Ipp64f** ppDst, int dstRoiShift,
    int dstStride0, int widthHeight, int count);

IppStatus ippmLUDecomp_ma_32f_L(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, int* pDstIndex, Ipp32f** ppDst,
    int dstRoiShift, int dstStride1, int dstStride2, int widthHeight,
    int count);
```

```
IppStatus ippmLUDecomp_ma_64f_L(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, int* pDstIndex, Ipp64f** ppDst,
    int dstRoiShift, int dstStride1, int dstStride2, int widthHeight,
    int count);
```

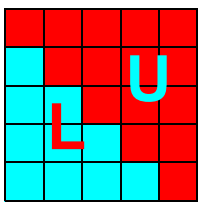
Parameters

pSrc, ppSrc	Pointer to the source matrix or array of matrices.
srcStride0	Stride between the matrices in the source array.
srcStride1	Stride between the rows in the source matrix(ces).
srcStride2	Stride between the elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).
pDstIndex	Pointer to array of pivot indices, where row *i* interchanges with row *index(i)*. The array size can be more than or equal to *widthHeight*. If the operation is performed on an array of matrices, the size of the array of indices must be more than or equal to *count*widthHeight*.
pDst, ppDst	Pointer to the destination matrix or array of matrices.
dstStride0	Stride between the matrices in the destination array.
dstStride1	Stride between the rows in the destination matrix.
dstStride2	Stride between the elements in the destination matrix.
dstRoiShift	ROI shift in the destination matrix.
widthHeight	Size of the square matrix.
count	Number of matrices in the array.

Description

The function `ippmLUDecomp` is declared in the `ippm.h` header file. The function represents the source matrix *pSrc* or *ppSrc* as a product of two matrices *L* and *U*, where *L* is the lower triangular with unit diagonal elements and *U* is the upper triangular (see Figure 6-1). Both *L* and *U* are stored in *pDst* or *ppDst*. Matrix elements located below the matrix diagonal are the lower triangular matrix *L*. The unit diagonal elements of the matrix *L* are not stored. The remaining matrix elements are the upper triangular matrix *U*. If necessary, the function implements the algorithm with partial pivoting that interchanges rows. Array of pivot indices is stored in *pDstIndex*.

Figure 6-1 LU Decomposition Matrix Storage



Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the size of the source matrix is equal to 0.
`ippStsSingularErr`	Returns an error when the source matrix is singular.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not divisible by size of data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the roiShift value is negative or not divisible by size of data type.
`ippStsCountMatrixErr`	Returns an error when the count value is less or equal to zero.

LUBackSubst

Solves system of linear equations with LU-factored square matrix.

Syntax

Case 1: Matrix - vector operation

```
IppStatus ippmLUBackSubst_mv_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, int* pSrcIndex, const Ipp32f* pSrc2, int src2Stride2,
    Ipp32f* pDst, int dstStride2, int widthHeight);
IppStatus ippmLUBackSubst_mv_64f(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, int* pSrcIndex, const Ipp64f* pSrc2, int src2Stride2,
    Ipp64f* pDst, int dstStride2, int widthHeight);
```

```
IppStatus ippmLUBackSubst_mv_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int* pSrcIndex, const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst,
    int dstRoiShift, int widthHeight);

IppStatus ippmLUBackSubst_mv_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int* pSrcIndex, const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst,
    int dstRoiShift, int widthHeight);
```

Case 2: Matrix - vector array operation

```
IppStatus ippmLUBackSubst_mva_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int* pSrcIndex, const Ipp32f* pSrc2,
    int src2Stride0, int src2Stride2, Ipp32f* pDst, int dstStride0,
    int dstStride2, int widthHeight, int count);

IppStatus ippmLUBackSubst_mva_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int* pSrcIndex, const Ipp64f* pSrc2,
    int src2Stride0, int src2Stride2, Ipp64f* pDst, int dstStride0,
    int dstStride2, int widthHeight, int count);

IppStatus ippmLUBackSubst_mva_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int* pSrcIndex, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp32f** ppDst, int dstRoiShift, int dstStride2, int widthHeight,
    int count);

IppStatus ippmLUBackSubst_mva_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int* pSrcIndex, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp64f** ppDst, int dstRoiShift, int dstStride0, int widthHeight,
    int count);

IppStatus ippmLUBackSubst_mva_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int* pSrcIndex, const Ipp32f** ppSrc2,
    int src2RoiShift, int src2Stride2, Ipp32f** ppDst, int dstRoiShift,
    int dstStride2, int widthHeight, int count);

IppStatus ippmLUBackSubst_mva_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int* pSrcIndex, const Ipp64f** ppSrc2,
    int src2RoiShift, int src2Stride2, Ipp64f** ppDst, int dstRoiShift,
    int dstStride2, int widthHeight, int count);
```

Case 3: Matrix array - vector array operation

```
IppStatus ippmLUBackSubst_mava_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int* pSrcIndex, const Ipp32f* pSrc2,
    int src2Stride0, int src2Stride2, Ipp32f* pDst, int dstStride0,
    int dstStride2, int widthHeight, int count);
```

```
IppStatus ippmLUBackSubst_mava_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, int* pSrcIndex, const Ipp64f* pSrc2,
    int src2Stride0, int src2Stride2, Ipp64f* pDst, int dstStride0,
    int dstStride2, int widthHeight, int count);
IppStatus ippmLUBackSubst_mava_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, int* pSrcIndex, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Stride0, Ipp32f** ppDst, int dstRoiShift, int dstStride0,
    int widthHeight, int count);
IppStatus ippmLUBackSubst_mava_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, int* pSrcIndex, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Stride0, Ipp64f** ppDst, int dstRoiShift, int dstStride0,
    int widthHeight, int count);
IppStatus ippmLUBackSubst_mava_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int* pSrcIndex, const Ipp32f** ppSrc2,
    int src2RoiShift, int src2Stride2, Ipp32f** ppDst, int dstRoiShift,
    int dstStride2, int widthHeight, int count);
IppStatus ippmLUBackSubst_mava_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, int* pSrcIndex, const Ipp64f** ppSrc2,
    int src2RoiShift, int src2Stride2, Ipp64f** ppDst, int dstRoiShift,
    int dstStride2, int widthHeight, int count);
```

Parameters

pSrc1, ppSrc1	Pointer to the source matrix or array of matrices. Matrix(ces) must be a result of calling LUDecomp.
src1Stride0	Stride between the matrices in the source matrix array.
src1Stride1	Stride between the rows in the source matrix(ces).
src1Stride2	Stride between the elements in the source matrix(ces).
src1RoiShift	ROI shift in the source matrix(ces).
pSrc2, ppSrc2	Pointer to the source vector or array of vectors.
src2Stride0	Stride between the vectors in the source vector array.
src2Stride2	Stride between the elements in the source vector(s).
src2RoiShift	ROI shift in the source vector(s).
pSrcIndex	Pointer to array of pivot indices. This array must be a result of calling LUDecomp. The array size can be more than or equal to *widthHeight*. If the operation is performed on an array of matrices, the size of the array of indices must be more than or equal to *count*widthHeight*.

pDst, ppDst	Pointer to the destination vector or array of vectors.
dstStride0	Stride between the vectors in the destination array.
dstStride2	Stride between the elements in the destination vector(s).
dstRoiShift	ROI shift in the destination vector(s).
widthHeight	Size of the square matrix, the source vector, and the destination vector.
count	Number of matrices and right-hand part vectors in the arrays.

Description

The function `ippmLUBackSubst` is declared in the `ippm.h` header file. The function solves for *x* the following systems of linear equations:

$A \cdot x = L \cdot U \cdot x = b$.

You should call the function `LUDecomp` to compute the LU decomposition of `A` before calling this function.

The following example demonstrates how to use the functions `ippmLUDecomp_m_32f` and `ippmLUBackSubst_mva_32f`. For more information, see also examples in the Getting Started chapter.

Example 6-1 ippmLUFactorization_32f

```
IppStatus LUFactorization_32f(void){
/* Source matrix with widthHeight=3 */
    Ipp32f pSrc[3*3] = { 3, -5, -10,
                        -1,  4,  2 ,
                         1, -2, -3 };
    int srcStride2 = sizeof(Ipp32f);
    int srcStride1 = 3*sizeof(Ipp32f);

    /* Solver right-part is 2 vectors with length=3 */
    Ipp32f pSrc2[3*2] = { 0,  2, 1,
                          1, -1, 2 };
    int src2Stride2 = sizeof(Ipp32f);
    int src2Stride0 = 3*sizeof(Ipp32f);
```

Example 6-1 ippmLUFactorization_32f (continued)

```
    Ipp32f pDecomp[3*3]; /* Decomposed matrix location */
    int decompStride2 = sizeof(Ipp32f);
    int decompStride1 = 3*sizeof(Ipp32f);

    Ipp32f pDst[3*2];    /* Solver destination location */
    int dstStride2 = sizeof(Ipp32f);
    int dstStride0 = 3*sizeof(Ipp32f);

    int pIndex[3];       /* Pivoting indeces location */
    int widthHeight = 3;
    int count = 2;

    IppStatus status = ippmLUDecomp_m_32f((const Ipp32f*)pSrc,
        srcStride1, srcStride2, pIndex,
        pDecomp, decompStride1, decompStride2, widthHeight);

    status = ippmLUBackSubst_mva_32f((const Ipp32f*)pDecomp,
        decompStride1, decompStride2, pIndex, pSrc2, src2Stride0,
        src2Stride2, pDst, dstStride0, dstStride2, widthHeight, count);

    printf_m_Ipp32f("LUDecomp result:", pDecomp, 3, 3, status);
    printf_v_int("Pivoting indeces:", pIndex, 3);
    printf_va_Ipp32f("2 destination vectors:", pDst, 3, 2, status);

    return status;
}
```

The program above produces the following output:

```
LUDecomp result:
3.000000  -5.000000  -10.000000
-0.333333  2.333333  -1.333333
0.333333  -0.142857  0.142857

Pivoting indeces:
0  1  2

2 destination vectors:
40.000000  6.000000  9.000000
47.000000  6.000000  11.000000
```

Return Values

ippStsOk	Returns no error.
ippStsNullPtrErr	Returns an error when at least one input pointer is NULL.
ippStsSizeErr	Returns an error when the size of the source matrix is equal to 0.
ippStsStrideMatrixErr	Returns an error when the stride value is not positive or not divisible by size of data type.
ippStsRoiShiftMatrixErr	Returns an error when the roiShift value is negative or not divisible by size of data type.
ippStsCountMatrixErr	Returns an error when the count value is less or equal to zero.

CholeskyDecomp

Performs Cholesky decomposition of a symmetric positive definite square matrix.

Syntax

Case 1: Matrix operation

```
IppStatus ippmCholeskyDecomp_m_32f(const Ipp32f* pSrc, int srcStride1,
    int srcStride2, Ipp32f* pDst, int dstStride1, int dstStride2,
    int widthHeight);
IppStatus ippmCholeskyDecomp_m_64f(const Ipp64f* pSrc, int srcStride1,
    int srcStride2, Ipp64f* pDst, int dstStride1, int dstStride2,
    int widthHeight);
IppStatus ippmCholeskyDecomp_m_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    Ipp32f** ppDst, int dstRoiShift, int widthHeight);
IppStatus ippmCholeskyDecomp_m_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    Ipp64f** ppDst, int dstRoiShift, int widthHeight);
```

Case 2: Matrix array operation

```
IppStatus ippmCholeskyDecomp_ma_32f(const Ipp32f* pSrc, int srcStride0,
    int srcStride1, int srcStride2, Ipp32f* pDst, int dstStride0,
    int dstStride1, int dstStride2, int widthHeight, int count);
```

```
IppStatus ippmCholeskyDecomp_ma_64f(const Ipp64f* pSrc, int srcStride0,
    int srcStride1, int srcStride2, Ipp64f* pDst, int dstStride0,
    int dstStride1, int dstStride2, int widthHeight, int count);
IppStatus ippmCholeskyDecomp_ma_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride2, Ipp32f** ppDst, int dstRoiShift, int dstStride2,
    int widthHeight, int count);
IppStatus ippmCholeskyDecomp_ma_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride2, Ipp64f** ppDst, int dstRoiShift, int dstStride2,
    int widthHeight, int count);
IppStatus ippmCholeskyDecomp_ma_32f_L(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, Ipp32f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int widthHeight, int count);
IppStatus ippmCholeskyDecomp_ma_64f_L(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, Ipp64f** ppDst, int dstRoiShift,
    int dstStride1, int dstStride2, int widthHeight, int count);
```

Parameters

pSrc, ppSrc	Pointer to the source matrix or array of matrices.
srcStride0	Stride between the matrices in the source array.
srcStride1	Stride between the rows in the source matrix(ces).
srcStride2	Stride between the elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).
pDst, ppDst	Pointer to the destination matrix or array of matrices.
dstStride0	Stride between the matrices in the destination array.
dstStride1	Stride between the rows in the destination matrix.
dstStride2	Stride between the elements in the destination matrix.
dstRoiShift	ROI shift in the destination matrix.
widthHeight	Size of the square matrix.
count	Number of matrices in the array.

Description

The function `ippmCholeskyDecomp` is declared in the `ippm.h` header file. The function performs Cholesky decomposition of the symmetric square matrix that is positive definite. The source matrix is represented as a product of two matrices L and L^T, where L is the lower triangular

and L^T is its transpose that can serve itself as the upper triangular. Result of the function operation is L matrix with inverse diagonal elements. The function uses only data in the lower triangular of the source matrix.

Return Values

ippStsOk	Returns no error.
ippStsNullPtrErr	Returns an error when at least one input pointer is NULL.
ippStsSizeErr	Returns an error when the input size parameter is equal to 0.
ippStsNotPosDefErr	Returns an error when the source matrix is not positive definite.
ippStsStrideMatrixErr	Returns an error when the stride value is not positive or not divisible by size of data type.
ippStsRoiShiftMatrixErr	Returns an error when the roiShift value is negative or not divisible by size of data type.
ippStsCountMatrixErr	Returns an error when the count value is less or equal to zero.

CholeskyBackSubst

Solves system of linear equations using the Cholesky triangular factor.

Syntax

Case 1: Matrix -vector operation

```
IppStatus ippmCholeskyBackSubst_mv_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp32f* pSrc2, int src2Stride2, Ipp32f* pDst,
    int dstStride2, int widthHeight);
IppStatus ippmCholeskyBackSubst_mv_64f(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp64f* pSrc2, int src2Stride2, Ipp64f* pDst,
    int dstStride2, int widthHeight);
IppStatus ippmCholeskyBackSubst_mv_32f_P(const Ipp32f** ppSrc1,
    int src1RoiShift, const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst,
    int dstRoiShift, int widthHeight);
IppStatus ippmCholeskyBackSubst_mv_64f_P(const Ipp64f** ppSrc1,
    int src1RoiShift, const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst,
    int dstRoiShift, int widthHeight);
```

Case 2: Matrix - vector array operation

```
IppStatus ippmCholeskyBackSubst_mva_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp32f* pSrc2, int src2Stride0, int src2Stride2,
    Ipp32f* pDst, int dstStride0, int dstStride2, int widthHeight, int count);

IppStatus ippmCholeskyBackSubst_mva_64f(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp64f* pSrc2, int src2Stride0, int src2Stride2,
    Ipp64f* pDst, int dstStride0, int dstStride2, int widthHeight, int count);

IppStatus ippmCholeskyBackSubst_mva_32f_P(const Ipp32f** ppSrc1,
    int src1RoiShift, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2,
    Ipp32f** ppDst, int dstRoiShift, int dstStride2, int widthHeight,
    int count);

IppStatus ippmCholeskyBackSubst_mva_64f_P(const Ipp64f** ppSrc1,
    int src1RoiShift, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2,
    Ipp64f** ppDst, int dstRoiShift, int dstStride2, int widthHeight,
    int count);

IppStatus ippmCholeskyBackSubst_mva_32f_L(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride2,
    Ipp32f** ppDst, int dstRoiShift, int dstStride2, int widthHeight,
    int count);

IppStatus ippmCholeskyBackSubst_mva_64f_L(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride2,
    Ipp64f** ppDst, int dstRoiShift, int dstStride2, int widthHeight,
    int count);
```

Case 3: Matrix array - vector array operation

```
IppStatus ippmCholeskyBackSubst_mava_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp32f* pSrc2, int src2Stride0,
    int src2Stride2, Ipp32f* pDst,  int dstStride0, int dstStride2,
    int widthHeight, int count);

IppStatus ippmCholeskyBackSubst_mava_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, const Ipp64f* pSrc2, int src2Stride0,
    int src2Stride2, Ipp64f* pDst, int dstStride0, int dstStride2,
    int widthHeight, int count);

IppStatus ippmCholeskyBackSubst_mava_32f_P(const Ipp32f** ppSrc1,
    int src1RoiShift, int src1Stride2, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Stride2, Ipp32f** ppDst, int dstRoiShift, int dstStride2,
    int widthHeight, int count);

IppStatus ippmCholeskyBackSubst_mava_64f_P(const Ipp64f** ppSrc1,
    int src1RoiShift, int src1Stride2, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Stride2, Ipp64f** ppDst, int dstRoiShift, int dstStride2,
    int widthHeight, int count);
```

```
IppStatus ippmCholeskyBackSubst_mava_32f_L(const Ipp32f** ppSrc1,
    int src1RoiShift, int src1Stride1, int src1Stride2, const Ipp32f** ppSrc2,
    int src2RoiShift, int src2Stride2, Ipp32f** ppDst, int dstRoiShift,
    int dstStride2, int widthHeight, int count);
IppStatus ippmCholeskyBackSubst_mava_64f_L(const Ipp64f** ppSrc1,
    int src1RoiShift, int src1Stride1, int src1Stride2, const Ipp64f** ppSrc2,
    int src2RoiShift, int src2Stride2, Ipp64f** ppDst, int dstRoiShift,
    int dstStride2, int widthHeight, int count);
```

Parameters

pSrc1, ppSrc1	Pointer to the source matrix or array of matrices. Must be a result of calling `CholeskyDecomp`.
src1Stride0	Stride between the matrices in the source array.
src1Stride1	Stride between the rows in the source matrix(ces).
src1Stride2	Stride between the elements in the source matrix(ces).
src1RoiShift	ROI shift in the source matrix(ces).
pSrc2, ppSrc2	Pointer to the source vector or array of vectors.
src2Stride0	Stride between the vectors in the source array.
src2Stride1	Stride between the elements in the source vector(s).
src2RoiShift	ROI shift in the source vector(s).
pDst, ppDst	Pointer to the destination vector or array of vectors.
dstStride0	Stride between the vectors in the destination array.
dstStride2	Stride between the elements in the destination vector(s).
dstRoiShift	ROI shift in the destination vector(s).
widthHeight	Size of the square matrix, the source vector, and the destination vector.
count	Number of matrices and right-hand part vectors in the arrays.

Description

The function `ippmCholeskyBackSubst` is declared in the `ippm.h` header file. The function solves for x the following systems of linear equations:

$A \cdot x = L \cdot L^{T} \cdot x = b$.

You should call the function `ippmCholeskyBackSubst` to perform Cholesky decomposition of *A* before calling this function.

The following example demonstrates how to use the functions `ippmCholeskyDecomp_m_32f` and `ippmCholeskyBackSubst_mva_32f`. For more information, see also examples in the Getting Started chapter.

Example 6-2 ippmCholesky_mva_32f

```
IppStatus cholesky_mva_32f(void){
    /* Source matrix with widthHeight=4 */
    Ipp32f pSrc[4*4] = { 10, 1,  2, 3,
                         1, 12,  4, 5,
                         2,  4, 13, 6,
                         3,  5,  6, 14 };

    int srcStride2 = sizeof(Ipp32f);
    int srcStride1 = 4*sizeof(Ipp32f);

    /* Solver right-part is 3 vectors with length=4 */
    Ipp32f pSrc2[3*4] = {1, 2,  3,  4,
                         5, 6,  7,  8,
                         9, 10, 11, 12 };
    int src2Stride2 = sizeof(Ipp32f);
    int src2Stride0 = 4*sizeof(Ipp32f);

    Ipp32f pDecomp[4*4] = {0}; /* Decomposed matrix location */
    int decompStride2 = sizeof(Ipp32f);
    int decompStride1 = 4*sizeof(Ipp32f);

    Ipp32f pDst[3*4];          /* Solver destination location */
    int dstStride2 = sizeof(Ipp32f);
    int dstStride0 = 4*sizeof(Ipp32f);

    int widthHeight = 4;
    int count = 3;
```

Example 6-2 ippmCholesky_mva_32f (continued)

```
    IppStatus status = ippmCholeskyDecomp_m_32f((const Ipp32f*)pSrc,
        srcStride1, srcStride2, pDecomp, decompStride1,
        decompStride2, widthHeight);

    status = ippmCholeskyBackSubst_mva_32f((const Ipp32f*)pDecomp,
        decompStride1, decompStride2, pSrc2, src2Stride0, src2Stride2,
        pDst, dstStride0, dstStride2, widthHeight, count);

    printf_m_Ipp32f("Cholesky decomposition:", pDecomp, 4, 4, status);
    printf_va_Ipp32f("3 destination vectors:", pDst, 4, 3, status);

    return status;
}
```

The program above produces the following output:

```
Cholesky decomposition:
0.316228  0.000000  0.000000  0.000000
0.316228  0.289886  0.000000  0.000000
0.632456  1.101565  0.296349  0.000000
0.948683  1.362462  1.155513  0.317685

3 destination vectors:
0.006629  0.034783  0.116639  0.221883
0.332266  0.260316  0.273350  0.290109
0.657903  0.485848  0.430061  0.358335
```

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when at least one input pointer is `NULL`.
`ippStsSizeErr`	Returns an error when the size of the source matrix is 0.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not divisible by size of data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the roiShift value is negative or not divisible by size of data type.
`ippStsCountMatrixErr`	Returns an error when the count value is less or equal to zero.

Least Squares Problem Functions

7

This chapter describes Intel® Integrated Performance Primitives (Intel® IPP) functions for small matrices that compute the matrix QR decomposition and solve the least squares (LS) problem to an overdetermined system of linear equations. A typical least-squares problem is as follows: given a matrix A and a vector b, find the vector x that minimizes the L2-norm $\|Ax - b\|^2$. The number of rows in matrix A is equal to *height* and the number of columns is equal to *width*, rank (A) = *width*.

Table 7-1 Least Squares Problem functions

Function Base Name	Operation
QRDecomp	Computes the QR decomposition for the given matrix.
QRBackSubst	Solves least squares problem for QR-decomposed matrix.

QRDecomp

Computes the QR decomposition for the given matrix.

Syntax

Case 1: Matrix operation

```
IppStatus ippmQRDecomp_m_32f(const Ipp32f* pSrc, int srcStride1,
    int srcStride2, Ipp32f* pBuffer, Ipp32f* pDst, int dstStride1,
    int dstStride2, int width, int height);
```

```
IppStatus ippmQRDecomp_m_64f(const Ipp64f* pSrc, int srcStride1,
    int srcStride2, Ipp64f* pBuffer, Ipp64f* pDst, int dstStride1,
    int dstStride2, int width, int height);
IppStatus ippmQRDecomp_m_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    Ipp32f* pBuffer, Ipp32f** ppDst, int dstRoiShift, int width, int height);
IppStatus ippmQRDecomp_m_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    Ipp64f* pBuffer, Ipp64f** ppDst, int dstRoiShift, int width, int height);
```

Case 2: Matrix array operation

```
IppStatus ippmQRDecomp_ma_32f(const Ipp32f* pSrc, int srcStride0,
    int srcStride1, int srcStride2, Ipp32f* pBuffer, Ipp32f* pDst,
    int dstStride0, int dstStride1, int dstStride2, int width, int height,
    int count);
IppStatus ippmQRDecomp_ma_64f(const Ipp64f* pSrc, int srcStride0,
    int srcStride1, int srcStride2, Ipp64f* pBuffer, Ipp64f* pDst,
    int dstStride0, int dstStride1, int dstStride2, int width, int height,
    int count);
IppStatus ippmQRDecomp_ma_32f_P(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp32f* pBuffer, Ipp32f** ppDst, int dstRoiShift,
    int dstStride0, int width, int height, int count);
IppStatus ippmQRDecomp_ma_64f_P(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride0, Ipp64f* pBuffer, Ipp64f** ppDst, int dstRoiShift,
    int dstStride0, int width, int height, int count);
IppStatus ippmQRDecomp_ma_32f_L(const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, Ipp32f* pBuffer, Ipp32f** ppDst,
    int dstRoiShift, int dstStride1, int dstStride2, int width, int height,
    int count);
IppStatus ippmQRDecomp_ma_64f_L(const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride1, int srcStride2, Ipp64f* pBuffer, Ipp64f** ppDst,
    int dstRoiShift, int dstStride1, int dstStride2, int width, int height,
    int count);
```

Parameters

pSrc, ppSrc	Pointer to the source matrix or array of matrices.
srcStride0	Stride between the matrices in source array.
srcStride1	Stride between the rows in the source matrix(ces).
srcStride2	Stride between the elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).

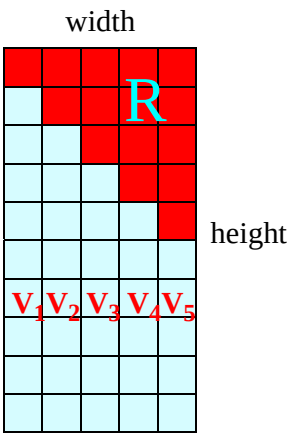
pBuffer	Pointer to a pre-allocated auxiliary array to be used for internal computations. The number of elements in the array must be at least *height*.
pDst, ppDst	Pointer to the destination matrix or array of matrices.
dstStride0	Stride between the matrices in the destination array.
dstStride1	Stride between the rows in the destination matrix.
dstStride2	Stride between the elements in the destination matrix.
dstRoiShift	ROI shift in the destination matrix.
width	Matrix width.
height	Matrix height.
count	Number of matrices in the array.

Description

The function ippmQRDecomp is declared in the ippm.h header file. The function computes the QR decomposition of a general matrix *A* with the number of rows *height* and the number of columns *width*, where *height* is more or equal to *width*,without the use of pivoting.

The function does not form the matrix *Q* explicitly. Instead, *Q* is represented as a cumulative product of *width* Householder vectors. Matrix elements located below the matrix diagonal are the Householder vectors v_n. The first components of these vectors are not stored because they are equal to 1. The remaining matrix elements are the upper triangular matrix *R* (see Figure 7-1).

Figure 7-1 QR Decomposition Matrix Storage



Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when one of the input pointers is `NULL`.
`ippStsSizeErr`	Returns an error when the size of the source matrix is equal to 0.
`ippStsDivByZeroErr`	Returns an error when the source matrix has an incomplete column rank.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not divisible by size of data type.
`ippStsRoiShiftMatrixErr`	Returns an error when the roiShift value is negative or not divisible by size of data type.
`ippStsCountMatrixErr`	Returns an error when the count value is less or equal to zero.
`ippStsSizeMatchMatrixErr`	Returns an error when the sizes of the source matrices are unsuitable.

QRBackSubst

Solves least squares problem for QR-decomposed matrix.

Syntax

Case 1: Matrix - vector operation

```
IppStatus ippmQRBackSubst_mv_32f(const Ipp32f* pSrc1, int src1Stride1,
    int src1Stride2, Ipp32f* pBuffer, const Ipp32f* pSrc2, int src2Stride2,
    Ipp32f* pDst, int dstStride2, int width, int height);

IppStatus ippmQRBackSubst_mv_64f(const Ipp64f* pSrc1, int src1Stride1,
    int src1Stride2, Ipp64f* pBuffer, const Ipp64f* pSrc2, int src2Stride2,
    Ipp64f* pDst, int dstStride2, int width, int height);

IppStatus ippmQRBackSubst_mv_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    Ipp32f* pBuffer, const Ipp32f** ppSrc2, int src2RoiShift, Ipp32f** ppDst,
    int dstRoiShift, int width, int height);

IppStatus ippmQRBackSubst_mv_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    Ipp64f* pBuffer, const Ipp64f** ppSrc2, int src2RoiShift, Ipp64f** ppDst,
    int dstRoiShift, int width, int height);
```

Case 2: Matrix - vector array operation

```
IppStatus ippmQRBackSubst_mva_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, Ipp32f* pBuffer, const Ipp32f* pSrc2,
    int src2Stride0, int src2Stride2, Ipp32f* pDst, int dstStride0,
    int dstStride2, int width, int height, int count);

IppStatus ippmQRBackSubst_mva_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, Ipp64f* pBuffer, const Ipp64f* pSrc2,
    int src2Stride0, int src2Stride2, Ipp64f* pDst, int dstStride0,
    int dstStride2, int width, int height, int count);

IppStatus ippmQRBackSubst_mva_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    Ipp32f* pBuffer, const Ipp32f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp32f** ppDst, int dstRoiShift, int dstStride0, int width, int height,
    int count);
```

```
IppStatus ippmQRBackSubst_mva_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    Ipp64f* pBuffer, const Ipp64f** ppSrc2, int src2RoiShift, int src2Stride0,
    Ipp64f** ppDst, int dstRoiShift, int dstStride0, int width, int height,
    int count);
IppStatus ippmQRBackSubst_mva_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, Ipp32f* pBuffer, const Ipp32f** ppSrc2,
    int src2RoiShift, int src2Stride2, Ipp32f** ppDst, int dstRoiShift,
    int dstStride2, int width, int height, int count);
IppStatus ippmQRBackSubst_mva_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, Ipp64f* pBuffer, const Ipp64f** ppSrc2,
    int src2RoiShift, int src2Stride2, Ipp64f** ppDst, int dstRoiShift,
    int dstStride2, int width, int height, int count);
```

Case 3: Matrix array - vector array operation

```
IppStatus ippmQRBackSubst_mava_32f(const Ipp32f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, Ipp32f* pBuffer, const Ipp32f* pSrc2,
    int src2Stride0, int src2Stride2, Ipp32f* pDst, int dstStride0,
    int dstStride2, int width, int height, int count);
IppStatus ippmQRBackSubst_mava_64f(const Ipp64f* pSrc1, int src1Stride0,
    int src1Stride1, int src1Stride2, Ipp64f* pBuffer, const Ipp64f* pSrc2,
    int src2Stride0, int src2Stride2, Ipp64f* pDst, int dstStride0,
    int dstStride2, int width, int height, int count);
IppStatus ippmQRBackSubst_mava_32f_P(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride0, Ipp32f* pBuffer, const Ipp32f** ppSrc2, int src2RoiShift,
    int src2Stride0, Ipp32f** ppDst, int dstRoiShift, int dstStride0, int width,
    int height, int count);
IppStatus ippmQRBackSubst_mava_64f_P(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride0, Ipp64f* pBuffer, const Ipp64f** ppSrc2, int src2RoiShift,
    int src2Stride0, Ipp64f** ppDst, int dstRoiShift, int dstStride0, int width,
    int height, int count);
IppStatus ippmQRBackSubst_mava_32f_L(const Ipp32f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, Ipp32f* pBuffer, const Ipp32f** ppSrc2,
    int src2RoiShift, int src2Stride2, Ipp32f** ppDst, int dstRoiShift,
    int dstStride2, int width, int height, int count);
IppStatus ippmQRBackSubst_mava_64f_L(const Ipp64f** ppSrc1, int src1RoiShift,
    int src1Stride1, int src1Stride2, Ipp64f* pBuffer, const Ipp64f** ppSrc2,
    int src2RoiShift, int src2Stride2, Ipp64f** ppDst, int dstRoiShift,
    int dstStride2, int width, int height, int count);
```

Parameters

pSrc1, ppSrc1	Pointer to the source matrix or array of matrices.

src1Stride0	Stride between the matrices in the source matrix array.
src1Stride1	Stride between the rows in the source matrix(ces).
src1Stride2	Stride between the elements in the source matrix(ces).
src1RoiShift	ROI shift in the source matrix(ces).
pSrc2, ppSrc2	Pointer to the source vector or array of vectors.
src2Stride0	Stride between the vectors in the source vector array.
src2Stride2	Stride between the elements in the source vector(s).
src2RoiShift	ROI shift in the source vector(s).
pBuffer	Pointer to a pre-allocated auxiliary array to be used for internal computations. The number of elements in the array must be at least *height*.
pDst, ppDst	Pointer to the destination matrix or array of matrices.
dstStride0	Stride between the vectors in the destination array.
dstStride2	Stride between the elements of the destination vector(s).
dstRoiShift	ROI shift in the destination vector(s).
width	Matrix width and destination vector length.
height	Matrix height and source vector length.
count	Number of matrices and right-hand part vectors in the array.

Description

The function `ippmQRBackSubst` is declared in the `ippm.h` header file. The function solves the least squares problem

$$\|A \cdot x - b\|_2^2 = \|R \cdot x - c\|_2^2 + \|d\|_2^2$$

where A is a matrix of linear equations system $A \cdot x = b$, x is the unknown variable vector, b is the vector of the right-hand equation system, R is the upper triangular matrix (see Figure 7-1), and

$$Q^T \cdot b = \begin{bmatrix} c \\ d \end{bmatrix}$$

where Q is the orthogonal matrix.

The number of equations in the system *height* is equal to or more than the number of unknown variables *width*.

You should call the function QRDecomp to compute the QR decomposition of A before calling this function.

The following example demonstrates how to use the functions ippmQRDecomp_m_32f and ippmQRBackSubst_mva_32f. For more information, see also examples in the Getting Started chapter.

Example 7-1 ippmQRFactorization_32f

```
IppStatus QRFactorization_32f(void){
    /* Source matrix with width=4 and height=5 */
    Ipp32f pSrc[5*4] = {1, 1, 1, 1,
                        1, 3, 1, 1,
                        1,-1, 3, 1,
                        1, 1, 1, 3,
                        1, 1, 1, -1 };
    int srcStride2 = sizeof(Ipp32f);
    int srcStride1 = 4*sizeof(Ipp32f);

    /* Solver right-part is 2 vectors with length=5 */
    Ipp32f pSrc2[2*5] = { 2, 1, 6, 3, 1,
                          3, 4, 5, 6, 1 };
    int src2Stride2 = sizeof(Ipp32f);
    int src2Stride0 = 5*sizeof(Ipp32f);
```

Example 7-1 ippmQRFactorization_32f (continued)

```
    Ipp32f pDecomp[5*4]; /* Decomposed matrix location */
    int decompStride2 = sizeof(Ipp32f);
    int decompStride1 = 4*sizeof(Ipp32f);

    Ipp32f pDst[2*4]; /* Solver destination location */
    int dstStride2 = sizeof(Ipp32f);
    int dstStride0 = 4*sizeof(Ipp32f);

    int width  = 4;
    int height = 5;
    int count  = 2;

    Ipp32f pBuffer[5];  /* Buffer location */

    IppStatus status = ippmQRDecomp_m_32f((const Ipp32f*)pSrc,
        srcStride1, srcStride2, pBuffer,
        pDecomp, decompStride1, decompStride2, width, height);

    status = ippmQRBackSubst_mva_32f((const Ipp32f*)pDecomp,
       decompStride1, decompStride2, pBuffer, pSrc2, src2Stride0,
       src2Stride2, pDst, dstStride0, dstStride2, width, height, count);

    printf_m_Ipp32f("QRDecomp result:", pDecomp, 4, 5, status);
    printf_va_Ipp32f("3 destination vectors:", pDst, 4, 2, status);

    return status;
}
```

The program above produces the following output:

```
QRDecomp result:
-2.236068 -2.236068  -3.130495  -2.236068
0.309017  -2.828427   1.414214  -0.000000
0.309017  -0.414214  -1.095445  0.000000
0.309017  -0.000000  -0.130449  -2.828427
0.309017  -0.000000  -0.130449  -0.414214

3 destination vectors:
0.500000  -0.500000   1.500000   0.500000
5.250001  -0.500000  -0.500001  -0.250000
```

Return Values

`ippStsOk`	Returns no error.
`ippStsNullPtrErr`	Returns an error when one of the input pointers is `NULL`.
`ippStsSizeErr`	Returns an error when the size of the source matrix is equal to 0.
`ippStsStrideMatrixErr`	Returns an error when the stride value is not positive or not divisible by size of data type.
`ippStsRoiShiftMatrixErr`	RoiShift value is negative or not divisible by size of data type.
`ippStsCountMatrixErr`	Returns an error when the count value is less or equal to zero.
`ippStsSizeMatchMatrixErr`	Returns an error when the sizes of the source matrices are unsuitable.

Eigenvalue Problem Functions

8

This chapter describes Intel® Integrated Performance Primitives (Intel® IPP) functions for small matrices that compute eigenvalues and eigenvectors for real symmetric matrices.

Table 8-1 Eigenvalue Problem functions

Function Base Name	Operation
EigenValuesVectorsSym	Finds eigenvalues and eigenvectors for real symmetric matrices (solves symmetric eigenvalue problem).
EigenValuesSym	Finds eigenvalues for real symmetric matrices.

EigenValuesVectorsSym

Finds eigenvalues and eigenvectors for real symmetric matrices (solves symmetric eigenvalue problem).

Syntax

Case 1: Matrix operation

```
IppStatus ippmEigenValuesVectorsSym_m_32f (const Ipp32f* pSrc, int srcStride1,
   int  srcStride2, Ipp32f* pBuffer, Ipp32f* pDstVectors, int dstStride1, int
   dstStride2, Ipp32f* pDstValues, int widthHeight);
IppStatus ippmEigenValuesVectorsSym_m_64f (const Ipp64f* pSrc, int srcStride1,
   int  srcStride2, Ipp64f* pBuffer,  Ipp64f* pDstVectors, int dstStride1, int
   dstStride2, Ipp64f* pDstValues, int widthHeight);
```

```
IppStatus ippmEigenValuesVectorsSym_m_32f_P (const Ipp32f** ppSrc, int
    srcRoiShift, Ipp32f* pBuffer, Ipp32f** ppDstVectors, int dstRoiShift,
    Ipp32f* pDstValues, int widthHeight);
IppStatus ippmEigenValuesVectorsSym_m_64f_P (const Ipp64f** ppSrc, int
    srcRoiShift, Ipp64f* pBuffer, Ipp64f** ppDstVectors, int dstRoiShift,
    Ipp64f* pDstValues, int widthHeight);
```

Case 2: Matrix array operation

```
IppStatus ippmEigenValuesVectorsSym_ma_32f (const Ipp32f* pSrc, int srcStride0,
    int srcStride1, int srcStride2, Ipp32f* pBuffer, Ipp32f* pDstVectors, int
    dstStride0, int dstStride1, int dstStride2, Ipp32f* pDstValues, int
    widthHeight, int count);
IppStatus ippmEigenValuesVectorsSym_ma_32f_P (const Ipp32f** ppSrc, int
    srcRoiShift, int srcStride0, Ipp32f* pBuffer, Ipp32f** ppDstVectors, int
    dstRoiShift, int dstStride0, Ipp32f* pDstValues, int widthHeight, int
    count);
IppStatus ippmEigenValuesVectorsSym_ma_32f_L (const Ipp32f** ppSrc, int
    srcRoiShift, int srcStride1,  int srcStride2, Ipp32f* pBuffer,  Ipp32f**
    ppDstVectors,  int dstRoiShift, int dstStride1, int dstStride2, Ipp32f*
    pDstValues, int widthHeight, int count);
IppStatus ippmEigenValuesVectorsSym_ma_64f (const Ipp64f* pSrc, int srcStride0,
    int srcStride1, int  srcStride2, Ipp64f* pBuffer, Ipp64f* pDstVectors, int
    dstStride0, int dstStride1, int dstStride2, Ipp64f* pDstValues, int
    widthHeight, int count);
IppStatus ippmEigenValuesVectorsSym_ma_64f_P (const Ipp64f** ppSrc, int
    srcRoiShift, int srcStride0, Ipp64f* pBuffer, Ipp64f** ppDstVectors, int
    dstRoiShift, int dstStride0, Ipp64f* pDstValues, int widthHeight, int
    count);
IppStatus ippmEigenValuesVectorsSym_ma_64f_L (const Ipp64f** ppSrc, int
    srcRoiShift, int srcStride1, int srcStride2, Ipp64f* pBuffer, Ipp64f**
    ppDstVectors, int dstRoiShift, int dstStride1, int dstStride2, Ipp64f*
    pDstValues, int widthHeight, int count);
```

Parameters

pSrc, *ppSrc*	Pointer to the source matrix or array of matrices.
srcStride0	Stride between matrices in the source array.
srcStride1	Stride between rows in the source matrix(ces).
srcStride2	Stride between elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).

pBuffer	Pointer to a pre-allocated auxiliary array to be used for internal computations. The number of elements in the array must be at least *widthHeight*^{2}.
pDstVectors,	
ppDstVectors	Pointer to the destination matrix or array of matrices whose columns are eigenvectors.
dstStride0	Stride between matrices in the destination array.
dstStride1	Stride between rows in the destination matrix.
dstStride2	Stride between elements in the destination matrix.
dstRoiShift	ROI shift in the destination matrix.
pDstValues	Pointer to the destination dense array that contains eigenvalues. The number of elementes in the array must be at least *widthHeight* for a matrix and *widthHeight*count* for an array of matrices.
widthHeight	Size of the square matrix.
count	The number of matrices in the array.

Description

The function `ippmEigenValuesVectorsSym` is declared in the `ippm.h` header file.

The function solves the Symmetric Eigenvalue Problem, that is finds the eigenvalues λ and corresponding eigenvectors $z \neq 0$ such that

$$Az=\lambda z ,$$

where A is a real symmetric square matrix of size *widthHeight*.

In case of a real symmetric matrix, all the *widthHeight* eigenvalues are real and there exists an orthonormal system of *widthHeight* eigenvectors. When all eigenvalues and eigenvectors are computed, the classical spectral factorization of A is

$$A=Z\Lambda Z^T ,$$

where Λ is a diagonal matrix whose non-zero elements are the eigenvalues, Z is an orthogonal matrix whose columns are the eigenvectors, and Z^T is its transpose.

The function stores eigenvalues in the array pointed by *pDstValues* densely and in the decreasing order. Eigenvectors of a source matrix are placed in columns of the appropriate destination matrix, pointed by *pDstVectors* or *ppDstVectors*.

The function uses only data in the lower triangular part of a source matrix **pSrc* or **ppSrc*.

The following example demonstrates how to use the function `ippmEigenValuesVectorsSym_m_32f`. For more information, see also examples in the Getting Started chapter.

Example 8-1 ippmEigenValuesVectorsSym_m_32f

```
IppStatus eigen_problem_32fvoid){
    /* Source matrix with width=4 and height=4 */
    Ipp32f pSrc[4*4]= {1, 1, 1, 3,
                       1, 3, 1, 3,
                       1, 1, 3, 1,
                       3, 3, 1, 3};
    int srcStride2 = sizeof(Ipp32f);
    int srcStride1 = 4*sizeof(Ipp32f);

    Ipp32f pBuffer[4*4];  /* Buffer location */
    int widthHeight = 4;

    Ipp32f pDstValues[4]; /* Eigen values location */

    Ipp32f pDstVectors[4*4]; /* Eigen vectors location */
    int dstStride2 = sizeof(Ipp32f);
    int dstStride1 = 4*sizeof(Ipp32f);

    IppStatus status=ippmEigenValuesVectorsSym_m_32f((const Ipp32f*)pSrc,
        srcStride1, srcStride2, pBuffer,
        pDstVectors,  dstStride1, dstStride2, pDstValues, widthHeight);

    printf_va_Ipp32f("Eigen values:", pDstValues, 4, 1, status);
    printf_m_Ipp32f("Eigen vectors:", pDstVectors, 4, 4, status);

   return status;
}
```

The program above produces the following output:

```
Eigen values:
7.911653  2.443112  1.121464  -1.476230
Eigen vectors:
-0.409522  -0.040703  -0.587074  -0.697122
-0.548069  -0.224421  0.749409  -0.296043
-0.327626  0.938908  0.071989  0.077017
-0.651593  -0.257742  -0.297569  0.648420
```

Return Values

`ippStsOk`	Indicates no errors.
`ippStsNullPtrErr`	Indicates an error if at least one input pointer is NULL.
`ippStsSizeErr`	Indicates an error if the input size parameter is less or equal to 0.
`ippStsStrideMatrixErr`	Indicates an error if a stride value is not positive or not divisible by the size of data type.
`ippStsRoiShiftMatrixErr`	Indicates an error if a roiShift value is negative or not divisible by the size of data type.
`ippStsCountMatrixErr`	Indicates an error when the count value is less or equal to 0.
`ippStsSingularErr`	Indicates an error if any of the input matrices is singular.
`ippStsConvergeErr`	Indicates an error if the algorithm does not converge.

EigenValuesSym

Find eigenvalues for real symmetric matrices.

Syntax

Case 1: Matrix operation

```
IppStatus ippmEigenValuesSym_m_32f (const Ipp32f* pSrc, int srcStride1, int
    srcStride2,  Ipp32f* pBuffer, Ipp32f* pDstValues, int widthHeight);
IppStatus ippmEigenValuesSym_m_64f (const Ipp64f* pSrc, int srcStride1, int
    srcStride2, Ipp64f* pBuffer, Ipp64f* pDstValues, int widthHeight);
IppStatus ippmEigenValuesSym_m_32f_P (const Ipp32f** ppSrc, int srcRoiShift,
    Ipp32f* pBuffer, Ipp32f* pDstValues, int widthHeight);
IppStatus ippmEigenValuesSym_m_64f_P (const Ipp64f** ppSrc,  int srcRoiShift,
    Ipp64f* pBuffer, Ipp64f* pDstValues, int widthHeight);
```

Case 2: Matrix array operation

```
IppStatus ippmEigenValuesSym_ma_32f (const Ipp32f* pSrc, int srcStride0, int
    srcStride1, int  srcStride2, Ipp32f* pBuffer, Ipp32f* pDstValues, int
    widthHeight, int count);
IppStatus ippmEigenValuesSym_ma_32f_P (const Ipp32f** ppSrc,  int srcRoiShift,
    int srcStride0, Ipp32f* pBuffer, Ipp32f* pDstValues,  int widthHeight, int
    count);
IppStatus ippmEigenValuesSym_ma_32f_L (const Ipp32f** ppSrc, int srcRoiShift,
    int srcStride1,  int  srcStride2,  Ipp32f* pBuffer, Ipp32f* pDstValues, int
    widthHeight, int count);
IppStatus ippmEigenValuesSym_ma_64f (const Ipp64f* pSrc, int srcStride0,    int
    srcStride1,  int  srcStride2,  Ipp64f* pBuffer,  Ipp64f* pDstValues, int
    widthHeight, int count);
IppStatus ippmEigenValuesSym_ma_64f_P (const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride0,  Ipp64f* pBuffer, Ipp64f* pDstValues, int widthHeight, int
    count);
IppStatus ippmEigenValuesSym_ma_64f_L (const Ipp64f** ppSrc, int srcRoiShift,
    int srcStride1,  int  srcStride2,  Ipp64f* pBuffer, Ipp64f* pDstValues, int
    widthHeight, int count);
```

Parameters

pSrc, *ppSrc*	Pointer to the source matrix or array of matrices.

srcStride0	Stride between matrices in the source array.
srcStride1	Stride between rows in the source matrix(ces).
srcStride2	Stride between elements in the source matrix(ces).
srcRoiShift	ROI shift in the source matrix(ces).
pBuffer	Pointer to a pre-allocated auxiliary array to be used for internal computations. The number of elements in the array must be at least $widthHeight^2$.
pDstValues	Pointer to the destination dense array that contains eigenvalues. The number of elements in the array must be at least *widthHeight* for a matrix and *widthHeight*count* for an array of matrices.
widthHeight	Size of the square matrix.
count	The number of matrices in the array.

Description

The function `ippmEigenValuesSym` is declared in the `ippm.h` header file.

The function finds eigenvalues for a real symmetric matrix A of size *widthHeight*. Each eigenvalue λ is a scalar such that

$$Az=\lambda z ,$$

for a vector $z \neq 0$ called an eigenvector. In case of a real symmetric matrix, all the *widthHeight* eigenvalues are real. The function stores eigenvalues in the array pointed by *pDstValues* densely and in the decreasing order.

The function uses only data in the lower triangular part of a source matrix **pSrc* or **ppSrc*.

The following example demonstrates how to use the function `ippmEigenValuesSym_ma_32f`. For more information, see also examples in the Getting Started chapter.

Example 8-2 ippmEigenValuesSym_ma_32f

```
IppStatus eigenvalues_ma_32f(void){
    /* Source data: 2 matrices with width=3 and height=3 */
    Ipp32f pSrc[2*3*3]= {1, 1, 1,
                         1, 3, 1,
                         1, 1, 3,
                         1, 1, 3,
                         1, 2, 1,
                         3, 1, 3};
    int srcStride2 = sizeof(Ipp32f);
    int srcStride1 = 3*sizeof(Ipp32f);
    int srcStride0 = 3*3*sizeof(Ipp32f);

    Ipp32f pBuffer[3*3];  /* Buffer location */
    int widthHeight = 3;
    int count = 2;

    Ipp32f pDstValues[2*3]; /* Eigen values location for two matrices */

    IppStatus status=ippmEigenValuesSym_ma_32f((const Ipp32f*)pSrc,
        srcStride0, srcStride1, srcStride2, pBuffer,
        pDstValues, widthHeight, count);

    printf_va_Ipp32f("Eigen values:", pDstValues, 3, 2, status);

    return status;
}
```

The program above produces the following output:

```
Eigen values:
4.561553  2.000000  0.438447
5.691268  1.488873  -1.180140
```

Return Values

`ippStsOk`	Indicates no errors.
`ippStsNullPtrErr`	Indicates an error if at least one input pointer is NULL.
`ippStsSizeErr`	Indicates an error if the input size parameter is less or equal to 0.
`ippStsStrideMatrixErr`	Indicates an error if a stride value is not positive or not divisible by the size of data type.
`ippStsRoiShiftMatrixErr`	Indicates an error if a roiShift value is negative or not divisible by the size of data type.
`ippStsCountMatrixErr`	Indicates an error when the count value is less or equal to 0.
`ippStsSingularErr`	Indicates an error if any of the input matrices is singular.
`ippStsConvergeErr`	Indicates an error if the algorithm does not converge.

Index

H

I

L

M